TEK REFERENCE MANUAL

Part No. 061-3393-00
Product Group 07

TEKTRONIX SMALLTALK

Please Check at the Rear of this Manual for NOTES and CHANGE INFORMATION

First Printing JUNE 1987

Tektronix
COMMITTED TO EXCELLENCE

Copyright © 1987, Tektronix, Inc. All rights reserved.

Tektronix products are covered by U.S. and foreign patents, issued and pending.

This document may not be copied in whole or in part, or otherwise reproduced except as specifically permitted under U.S. copyright law, without the prior written concent of Tektronix, Inc. P.O. Box 500, Beaverton, Oregon 97077.

Specifications subject to change.

TEKTRONIX ,TEK, and UTek are registered trademarks of Tektronix, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

PELLUCIDA is a trademark of Bigelow & Holmes.

HELVETICA and TIMES are registered trademarks of Linotype Corp.

UniFLEX is a registered trademark of Technical Systems Consultants, Inc.

Smalltalk-80 and XEROX are trademarks of Xerox Corporation.

Revision Information

MANUAL: Tektronix Smalltalk Reference

This manual supports the following versions of this product: Image Version: TB2.2.1

REV DATE	DESCRIPTION
JUNE 1987	Original Issue

Contents

Introduction

Classes

AbstractFileStatus
AbstractSystemCall
AimSystemCall
DisplayState
ExternalBinaryData
ExternalData
ExternalPointerData
FileDirectory
FileStream
FixedSizeExternalPointerData
Inaddr
IntegerPointer
Iovec
Itimerval
Ltchars
Msghdr
OSFilter
Pipe
PipeReadStream
PipeStream
PipeWriteStream
PointerArray
Rlimit
Rusage
ScreenView
Sgttyb
SockaddrIn
SockaddrUn
Stat
StrikeFont
StrikeFontManager
StructOutputTable
StructureArray
Subtask
Tchars
TextStyle
TextStyleManager
Timeval

Timezone
UniflexFileStatus
UniflexSystemCall
UTekFileStatus
UTekSystemCall
Utsname
VirtualStrikeFont
Wait
WorkspaceController

Introduction

This manual, as its name implies, is meant to be a source of reference information about Tektronix Smalltalk. Use this manual when you seek information about a class, such as its protocol, background to relate the class to other Smalltalk classes and the world outside Smalltalk, and examples of how to use the class. If you are new to Smalltalk or need a topical approach to a problem, you should read the *Tektronix Smalltalk Users* manual before, or in addition to, looking here. Since this manual does not cover all Smalltalk classes, you might want to refer to other Smalltalk texts.

The first edition of the *Tektronix Smalltalk Reference* manual contains Smalltalk classes added to the system by Tektronix and classes which appeared in Smalltalk-80 but have been substantially revised or enlarged by Tektronix. This edition includes 47 classes that fit those criteria for inclusion. Future editions of the manual will document classes meeting the same criteria as this edition, plus classes which have not been previously documented in print. Other information which has been proposed for inclusion in future editions includes

- Pool Dictionaries,
- Global Variables,
- Error Messages, and
- File Formats.

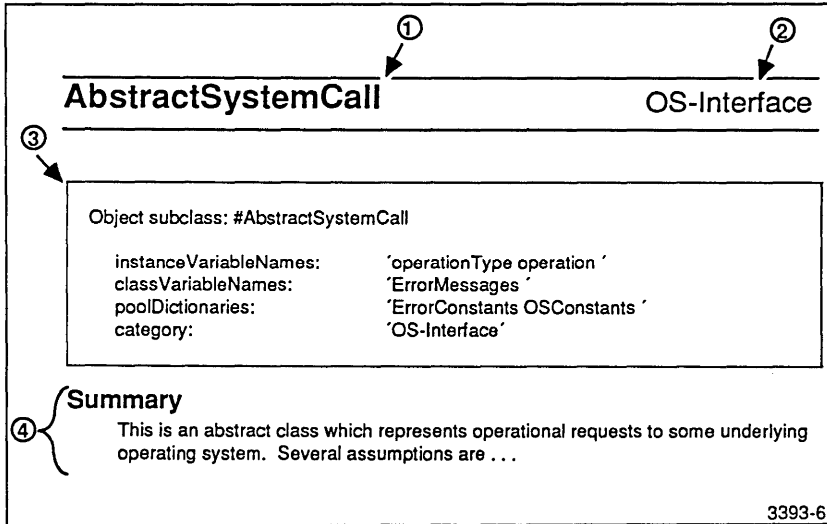
Arrangement of this Book

Following this introduction you will find the classes arranged alphabetically. The alphabetic arrangement of classes will continue in future editions, and additional sections for the information outlined above will follow the classes.

There are some classes in Tektronix Smalltalk images which are operating system specific. They only appear in an image released for a particular operating system. For example, **UniflexSystemCall** does not appear in the TB2.2.1 image released with the UTek operating system. The system dependency (e.g., *(UniFLEX only)*) is noted in the upper right corner of the class′ documentation in the manual. Classes in this release which are UTek-dependent are *not* indicated in that manner, because all users of this manual have the image for the UTek operating system.

Contents of Class Entries

The figure below is an example of the standard format which begins each class entry.



Partial Class Entry (Sample)

Explanation

①— class name

②— class category

③— class definition

④— class comment

Parts 1 through 4 of every entry contain information extracted directly from the image. If a class is operating system-dependent (or otherwise release-specific), that restriction is included in Part 2.

Additional Parts of Class Entries

After the Summary, class entries vary. The following is a complete list of sections which appear in various classes, in the order of their appearance.

Instance Variables

An alphabetic list of instance variables, their default class, and a brief description of each variable.

Inherited Instance Variables

Sometimes this is included to explain class-specific usage of inherited variables. It follows the format of the Instance Variables.

Class Variables

An alphabetic list of class variables, their default class, and a brief description of each variable. An exception to the usual format was made for concrete classes in the **ExternalData** hierarchy. The conventions established for defining their class variables are discussed under the appropriate superclass, **ExternalBinaryData** or **ExternalPointerData.**

Inherited Class Variables

Sometimes this is included to explain class-specific usage of inherited variables. It follows the format of the Class Variables.

Pool Dictionaries

An alphabetic list of shared pools and a brief description of each dictionary.

Inherited Pool Dictionaries

Sometimes this is included to explain class-specific usage of inherited shared pools. It follows the format of the Pool Dictionaries.

Instance Methods ———————————————

message category

messageSelector
method comment

NameOfClass class	
instanceVariableNames:	´name1 name2 etc.´

The box above only appears if a class has class instance variables.

NameOfClass class — Instance Variables

An alphabetic list of class instance variables, their default class, and a brief description of each variable.

Class Methods

message category

messageSelector
method comment

Rationale
The purpose of this class — why it's in the image.

Background
Information relating this class to other classes and, sometimes, to the operating system.

Discussion
Implementation information, a description of the class' protocol, practical advice.

Examples
"How to" information, sample code using the class and an explanation of the sample code. When the example is taken from the image, its class location is given.

Related Classes
Classes are listed if they would be helpful in the understanding of this class. The list might include subclasses (if any), other classes often used with this class, and the superclass. Classes are listed here even if they are not currently included in this manual.

Typeface and Font Conventions

Some conventions were established for the use of fonts and typefaces in the Summary, Rationale, Background, Discussion, Examples, and Related Classes sections of each entry. Different fonts and typefaces were usually *not* used in method comments, nor in the descriptions of instance variables, class variables, pool dictionaries, and class instance variables. Those parts are derived from the released image. Occasionally, an engineer has used the '#' symbol to denote a global variable in method or variable comments (e.g., . . . global variable #OS . . .).

- Code fragments and example code appear in a serif font, for example:

  Transcript show: . . .

- Information that you would type at the keyboard appears in a monospaced (typewriter-style) font, for example:

  . . . bring up the standard image by typing `smalltalk`, you . . .

- A bold face is used for the names of classes, instance and class variables, pool dictionaries, global variables, and message selectors.

  global variables:

  . . . install the appropriate subclass as **OS** . . .
  . . . path relative to **Disk** . . .

  class name and message selector:

  . . . using the **PositionableStream** method **nextCNumber** . . .

  instance variables and shared variables:

  . . . the value at **DefaultTextStyle** in the **TextConstants** dictionary . . . (**TextConstants** is a shared pool.)
  . . . deal with the **Forms** in the **glyphs** instance variable . . .
  . . . the class variable **BrkcDataIndex** holds the offset . . .
  . . . the **sizeInBytes** class instance variable . . .

- Italic is used for the names of UTek system calls and utilities, for file and path names, the names of protocols (also called "message categories"), and temporary variables in example code.

  UTek system calls and utilities:

  . . . access the UTek program *wc*.
  . . . some filters such as *cat*, which . . .
  . . . the *accept(2)* system call . . .

  book titles and sections of *UTek Command Reference*:

  . . . documented under *inet(4)* in the manual *UTek Command Reference, Volume 2*.

  temporary variables:

  . . . the three temporary variables *in*, *out*, and *err*

file names and path names:

. . . Examples are *BertrandVariable12.font* . . .
. . . structure is found in *sys/un.h.*

protocols (message categories):

. . . the *initialize-release* method **initializeFrom:**.

About Preliminary Manuals

The first time a manual is published it may contain some inaccuracies or be incomplete — this manual is considered a preliminary manual. Your comments and corrections are encouraged, so that the next edition of this manual will be an improvement and no longer be preliminary. In the case of this particular manual, the term "preliminary" is appropriate because important information that is expected in a reference manual has not been included. As explained above, new sections providing information in addition to the classes will appear in future editions.

What Do You Think?

Eventually, it would be great if this manual documented every class in Tektronix Smalltalk. That's a long-range plan. Until that time, you might want to make known your nominations for the classes or class categories that you want documented. Aside from that, write about your gripes (or even what you like!) and someone responsible for this documentation will respond — that's a promise.

Send your correspondence about this manual to:

Tektronix, Inc.
P.O. Box 500, M.S. 50-470
Beaverton, OR 97077
Attn: Tektronix Smalltalk Reference Manual

```
Object subclass: #AbstractFileStatus

    instanceVariableNames:      ''
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Interface'
```

Summary

AbstractFileStatus is an abstract class which represents the status of an operating system file. Instances of concrete subclasses of this class encapsulate information about a file status.

A file status is returned by the portable operations named

status:

and

statusName:

which are defined by the system call class for your operating system, referred to by the global variable **OS**.

Instance Methods

accessing

buffer
Answer the buffer which holds the file status information. *Subclass responsibility.*

description
Answer a String that contains a short description (file size and last modification time) of the receiver.

fileSize
Answer the file size in bytes. *Subclass responsibility.*

isDirectory
Answer whether the receiver describes a directory. *Subclass responsibility.*

isReadable
Answer true if the file represented by the receiver is readable. *Subclass responsibility.*

isWritable
Answer true if the file represented by the receiver is writable. *Subclass responsibility.*

lastModified
Answer the time of the last modification to the file. *Subclass responsibility.*

comparing

= aFileStatus
Answer true if the receiver and aFileStatus describe the same file. *Subclass responsibility.*

hash
Hash is reimplemented because = is implemented. *Subclass responsibility.*

copying

copy
Answer another instance just like the receiver.

Rationale

The operating system maintains many pieces of information about a file, such as read and write permissions. File status information is available directly through the file status class for your operating system, or indirectly through **FileStream**.

AbstractFileStatus establishes the pattern of protocol to be included in concrete subclasses for different operating systems. As an abstract class it provides a portable view of file status.

To obtain information about a file, the message selectors defined by this class are preferred. Methods are implemented in the appropriate subclass for your operating system. The subclass for your operating system may have additional methods to provide pieces of file status information unique to your operating system.

Related Classes

Subclasses:
UTekFileStatus
UniflexFileStatus

You might want to review the protocol for accessing file status information provided by **FileStream.** •

```
Object subclass: #AbstractSystemCall

    instanceVariableNames:    'operationType operation '
    classVariableNames:       'ErrorMessages '
    poolDictionaries:         'ErrorConstants OSConstants '
    category:                 'OS-Interface'
```

Summary

This is an abstract class which represents operational requests to some underlying operating system. Several assumptions are made about this operating system:

- It has multiple directories that contain files and/or other directories.
- It has files that have some form of status information and can be randomly read, written, and truncated.
- It has some means of running other programs. Multi-tasking is not assumed.
- It has some facility, such as pipes, for communicating with other running programs.
- It has file descriptors, which are used for uniquely identifying files and pipes.
- It has some means of receiving and sending interrupts.

Instances of concrete subclasses of this class encapsulate all information involved in making a single request to the operating system. This typically includes arguments, results, and control information.

System calls are performed by creating an instance containing the arguments for the call and then sending an "execution" message to the instance. Subclasses will typically define instance creation messages for all possible system calls supported by the host operating system.

The class protocol also defines a set of "portable" or generic operations. These high level functions are expected to be supported by any modern operating system. These operations are performed by sending a message directly to the class, which will cause the operations to be performed and the result returned. It is the responsibility of the subclasses to provide the system dependent implementation of these operations.

Wherever possible, the "portable" operations should be used to ensure operating system independence. The global variable **OS** is set to a subclass of **AbstractSystemCall**, as appropriate for the system. **OS** should be used in code (instead of the name of your system call class) to insure the portability of the code.

Instance Variables

operation <SmallInteger>
Operation code passed to the operating system.

operationType <Symbol>
Identifies which type of operating system interface (which primitive) this call uses.

Class Variables

ErrorMessages <Array>
An Array of error messages Strings indexed by the integer returned by the message errorCode.

Pool Dictionaries

ErrorConstants
Symbolic names are associated with error codes.

OSConstants
Symbolic names are associated with constants used by the various operating systems.

Instance Methods

constants

constant: aString
Return a value from the OSConstants or ErrorConstants pool that corresponds to aString. If aString is not found, notify the user that it is a bad key. This method need only be used for keys that are illegal Smalltalk identifiers (i.e., contain the underscore character), otherwise, the pool key name can be used directly by classes that declare the appropriate pool dictionaries.

errors

errorCode
Return an integer which identifies an error condition. This may not be valid if no error has occurred. *Subclass responsibility.*

errorCodeFor: errorString
Answer a value from the ErrorConstants pool that corresponds to aString. If aString is not found, notify the user that it is a bad key.

errorKeyword
Return a keyword associated with the present error condition.

errorKeywordFor: errorIndex
Answer a keyword from the ErrorConstants pool that corresponds to errorIndex. If none is found, return a String representation of the errorIndex.

errorString
Return a long, descriptive string describing the present error condition. If there is no string available, return aString representation of the error code.

issueError
Issue a notifier with a string that is associated with the present error code. If there is no string available, issue a notifier with the error code.

execution

environmentInvoke
Return various system information depending on the value of the operation instance variable. Possible values of operation are:

operation	system information
1	command line arguments' address
2	environment variables' address
3	hardware configuration block address (UTek only)
4	interpreter version string' address
5	copyright string address
6	OS and machine identification
7	SmallInteger time correction

invoke
Perform the system call, return true if it executed without error, return false otherwise. Users of #invoke must check the return value if they expect valid results! *Subclass responsibility.*

value
Evaluate the system call represented by the receiver. Create an error notifier if the system call results in an error.

valueIfError: aBlock
Evaluate the system call represented by the receiver. Evaluate aBlock if the system call resulted in an error.

operation type

environmentOperation
The desired operation involves an environment request.

Class Methods

class initialization

initialize
Initialize the pool dictionaries and class variables used by AbstractSystemCall and its subclasses. This should be overridden by a concrete subclass.

install
This message is normally executed by a concrete subclass of AbstractSystemCall, causing that subclass to become known by the global variable #OS.

whoAmI
This message attempts to determine which concrete subclass is a valid value for the global variable #OS.

constants

constant: aString
Return a value from the OSConstants or ErrorConstants pool that corresponds to aString. If aString is not found, notify the user that it is a bad key. This method need only be used for keys that are illegal Smalltalk identifiers (i.e., contain the underscore character), otherwise, the pool key name can be used directly.

errorKeywordFor: errorIndex

Return a String from the ErrorConstants pool that corresponds to errorIndex. If none is found, return a String representation of the errorIndex.

errorValueFor: errorString

Return a value from the ErrorConstants pool that corresponds to aString. If aString is not found, notify the user that it is a bad key. This method need only be used for keys that are illegal Smalltalk identifiers (i.e., contain the underscore character, for example, 'EDFS_CD'), otherwise, the variable names can be used directly.

keysAtValue: val

Return a SortedCollection of all OSConstants keywords that are associated with the value val.

keywordsForConstant: val

Return a SortedCollection of all OSConstants keywords that are associated with the value val.

file names

backupFileName: aFileName

Answer a string which is a backup file name for the file named, aFileName. *Subclass responsibility.*

checkFileName: aFileName **fixErrors:** fixErrors

Check aFileName for validity as a file name. If there are problems (e.g., illegal length or characters) and fixErrors is false, notify an error. If there are problems and fixErrors is true, make the name legal (by omitting illegal characters) and answer the new name. Otherwise, answer the name. *Subclass responsibility.*

completePathname: aDirectoryName

Answer the complete path name of the string, aDirectoryName, starting at the root of the file system. A trailing path name separator is considered part of a directory name even if not explicitly stored. *Subclass responsibility.*

currentDirectoryPseudonym
Answer the string which represents the name for the current directory. *Subclass responsibility.*

fileDirectory: aFileDirectory **directoryName:** aFileName
Answer a file directory name derived from the string, aFileName, and the file directory, aFileDirectory. *Subclass responsibility.*

fullDirectoryName: aDirectoryName
Answer the full path name of the string, aDirectoryName, starting at the directory Disk. A trailing path name separator is considered part of a directory name even if not explicitly stored. *Subclass responsibility.*

isBackupFileName: aFileName
Does aFileName correspond to a name that is usually a backup file name? *Subclass responsibility.*

isFileDirectoryName: aFileName
Is aFileName the name of a directory? *Subclass responsibility.*

isInvisibleFileName: aFileName
Is aFileName such that it would normally not be displayed in a file listing? *Subclass responsibility.*

separateDirectoryNameAndFileName: aFileName
Split the string, aFileName, into a directory name and a file name within the directory. Return an Array of two elements. Array at: 1 is the directory name, Array at: 2 is the file name. *Subclass responsibility.*

general inquiries

asTime: osSeconds
Convert the operating system's notion of time to a Time. *Subclass responsibility.*

defaultInterruptCode
Answer an operating system representation of the default interrupt action. *Subclass responsibility.*

fileStatusClass
Return the class used to store file status returned by system calls that return information about files. *Subclass responsibility.*

fontDirectory
Return the FileDirectory which contains font files. Each file contains a font in external font format. *Subclass responsibility.*

ignoreInterruptCode
Answer an operating system representation of the ignore interrupt action. *Subclass responsibility.*

isValid
Does this class represent the operating system running on this machine? (This method must return true in only one subclass.)

maxFileNameSize
Answer the maximum number of characters permissible in file names. *Subclass responsibility.*

maxOpenFiles
Answer the maximum number of files that may be open at one time. *Subclass responsibility.*

smalltalkInitializationDirectory
Return the FileDirectory which contains initialization files. Each file contains Smalltalk readable data used during class initialization. *Subclass responsibility.*

smalltalk InterpreterVersionString
Return a String identifying the Smalltalk interpreter that is currently running. *Subclass responsibility.*

textLineDelimiter
Answer a String containing the characters which delimit a 'line' in a file. *Subclass responsibility.*

portable directory operations

changeDirectory: directoryName
Change the current directory to the specified directory. *Subclass responsibility.*

createDirectory: directoryName
Create a new directory with the name directoryName. *Subclass responsibility.*

currentDirectoryName
Return a String with the name of the current working directory. *Subclass responsibility.*

nextFileName: directoryStream
Answer the next file name in directoryStream. Advance the directory stream beyond that name. Answer nil if none. Avoid recursion; do not return the current directory, even if it is next. *Subclass responsibility.*

removeDirectory: directoryName
Remove the directory named aDirectoryName. *Subclass responsibility.*

portable file operations

closeFile: aFileDescriptor
Close the file referred to by aFileDescriptor. *Subclass responsibility.*

create: aString
Create a new file named aString. Answer a writeOnly fileDescriptor for the file. *Subclass responsibility.*

existingName: fileName
Answer true if a file or directory named fileName exists. *Subclass responsibility.*

freeFileDescriptors
Answer the number of available file descriptors. For some operating systems this might be a very large number! *Subclass responsibility.*

open: fileName
Open the file named fileName. Answer a readWrite fileDescriptor for the file. *Subclass responsibility.*

openForRead: fileName
Open the file named fileName. Answer a readOnly fileDescriptor for the file. *Subclass responsibility.*

openForWrite: fileName
Open the file named fileName. Answer a writeOnly fileDescriptor for the file. *Subclass responsibility.*

read: fileDescriptor **into:** aStringOrByteArray
Fill aStringOrByteArray with data from the file referred to by fileDescriptor. Return the number of bytes read, or zero if at end. *Subclass responsibility.*

read: fileDescriptor **into:** aStringOrByteArray **size:** count
Fill aStringOrByteArray with, at most, count data elements from the file referred to by fileDescriptor. Return the number of bytes read, or zero if at end. *Subclass responsibility.*

remove: fileName
Remove the file named fileName. *Subclass responsibility.*

rename: fileName **as:** newFileName
Rename the file named fileName to have the name newFileName. Notify an error if fileName does not exist; but not if newFileName exists. *Subclass responsibility.*

seek: aFileDescriptor **to:** aFilePosition
Position the file known by aFileDescriptor to aFilePosition bytes from its beginning. *Subclass responsibility.*

shorten: fileDescriptor
Shorten a file to its current position. *Subclass responsibility.*

size: fd
Return the count of available bytes from the file or pipe known by the file descriptor fd. *Subclass responsibility.*

status: fileDescriptor
Answer a FileStatus for the file referred to by its fileDescriptor. *Subclass responsibility.*

statusName: fileName
Answer a FileStatus for the file named fileName. *Subclass responsibility.*

validFileDescriptor: fileDescriptor
Answer true if an open file with the specified file descriptor exists. *Subclass responsibility.*

write: fileDescriptor **from:** aStringOrByteArray **size:** byteCount
Write byteCount bytes of data from aStringOrByteArray to the file known by fileDescriptor. *Subclass responsibility.*

portable subtask operations

defaultInterrupt: anInterruptID
Set the specified interrupt to its default action.

executeUtility: aCommand **withArguments:** anOrderedCollection
Execute a binary program and return the entire results generated by the program as a string. No mechanism for input to the executable program is provided. Notify an error if the program cannot be executed or if the program terminates abnormally. *Subclass responsibility.*

executeUtilityWithErrorMapping: aCommand
withArguments: anOrderedCollection
Execute a binary program and return an array of two strings. The first string contains the entire normal output generated by the program. The second string contains any error message output from the program. No mechanism for input to the executable program is provided. Notify an error if the program cannot be executed or if the program terminates abnormally. *Subclass responsibility.*

ignoreInterrupt: anInterruptID
Set the specified interrupt to be ignored.

setInterrupt: interruptID **to:** aSemaphoreOrParameter
Override the default action for the interrupt known by interruptID by connecting it to a semaphore or some system specific parameter. If specified, the semaphore is posted on interrupt. The state of the interrupt once it has been received is system dependent. *Subclass responsibility.*

shell
Cause control to be passed to the operating system command interpreter. Return to Smalltalk is system dependent. *Subclass responsibility.*

Rationale

There was a desire to move system call dependencies from individual classes like **FileStream, Pipe,** and **Subtask** to one hierarchy and implement them there so that they are more easily maintained for different operating systems. A hierarchy was established to deal with multiple operating systems. **AbstractSystemCall** was created to define a minimum set of services to be provided to Smalltalk by any operating system. **AbstractSystemCall** defines those services with selector names that an application program can use and guarantee portability to any Tektronix Smalltalk operating system.

AbstractSystemCall establishes the pattern of protocol to be included in concrete subclasses. Some protocol for generic services is implemented in this class (instead of being a subclass responsibility), but much of it is reimplemented in subclasses. A concrete subclass of this class is assigned as the value of the global variable **OS.** In code, **OS** is used for portability, instead of the name of a system call class, unless the application is meant to be used only with a specific operating system.

The class variables are declared in **AbstractSystemCall** so that they are named in one place, and are initialized by separate system call classes.

Discussion

Since it is an abstract class, there is no instance creation protocol for this class. The protocol here is a template for subclasses. To make a system call, create an instance of a subclass. Read about the system call class for your operating system in this manual. It will implement the protocol found here, plus additional protocol which is operating system dependent.

Class Protocol

Class initialization methods intialize the class variables and pool dictionaries, install the appropriate subclass as **OS**, and determine which subclass should be **OS**. The **initialize** method returns self, because that method should be overridden by a subclass. You might have use for the **whoAmI** message in operating system dependent code. If you want to specify a certain operating system in the code, you could include something like

```
AbstractSystemCall whoAmI name = #ParticularSystemCall ifTrue:[ . . .
```

Use of the message **name** and the symbol are needed so that a correct boolean will be returned instead of a notifier that **ParticularSystemCall** is not recognized.

Constants methods access the pool dictionaries, **ErrorConstants** and **OSConstants**. Two methods, **constant:** and **errorValueFor:**, access a pool for an argument String which contains an underscore or other illegal identifier character. Another use for the **constant:** message is in a workspace, for example, where the pool dictionary is not directly accessible.

File names methods are implemented by a subclass.

General inquiries methods are implemented by a subclass.

Portable directory operations methods are implemented by a subclass.

Portable file operations methods are implemented by a subclass.

Portable subtask operations methods are implemented by a subclass, except for **defaultInterrupt:** and **ignoreInterrupt:**, which may be overridden by a subclass.

Instance Protocol

Constants has one method, **constant:**, which calls the class method of the same name.

Errors has several methods for accessing **ErrorConstants** and specifies one method, **errorCode**, to be implemented by a subclass. Some of the methods implemented here are overridden in a subclass.

Execution has several methods to execute a system call and one method, **environmentInvoke**, that executes the system call primitive for environment operations. Some of the methods implemented here are overridden in a subclass.

Operation type has one method to set the **operationType** instance variable for an environment operation.

Examples

The following code can be executed in a workspace. It will cause the name of your system call class to display in the System Transcript. There are no other messages that you are likely to send to this class.

```
Transcript cr; show: AbstractSystemCall whoAmI printString.
```

Related Classes

Subclass:

AimSystemCall

```
AbstractSystemCall subclass: #AimSystemCall

    instanceVariableNames:    'D0In D0Out D1In D1Out D2In A0In A0Out errno
                              '
    classVariableNames:       'StringTerminator '
    poolDictionaries:         ''
    category:                 'OS-Interface'
```

Summary

This is an abstract class which defines the basic mechanism used to perform operating system calls on the various Tektronix 4000 series bitmapped workstations. It also provides interfaces for accessing the display subsystem, operating system environment variables, and the command line arguments used when Smalltalk was invoked.

These system call objects contain all of the information normally used to perform a system call by an assembly language programmer. This includes the values of machine registers passed to or returned from the system call, any parameters lists, the operation ID of the function to be performed, and any error information. Since there may be several different interfaces to the underlying operating system, instances also contain a field which identifies which particular type of interface needs to be used. Input and output parameter descriptors are defined in subclasses.

Instance Variables

A0In <Object>
An input parameter descriptor for the value to be passed in A0.

A0Out <Object>
An output parameter descriptor for the value to be returned in A0.

D0In <Object>
An input parameter descriptor for the value to be passed in D0.

D0Out <Object>
An output parameter descriptor for the value to be returned in D0.

D1In <Object>
An input parameter descriptor for the value to be passed in D1.

D1Out <Object>
An output parameter descriptor for the value to be returned in D1.

D2In <Object>
An input parameter descriptor for the value to be passed in D2.

errno <SmallInteger or nil>
Error code return by the operating system, nil if no error.

Class Variables

StringTerminator <String>
This must be appended to strings before sending them to the operating system.

Instance Methods

initialize-release

D0In: din **D0Out:** dout **A0In:** ain **A0Out:** aout
Set D and A registers. Set unspecified input registers to nil and unspecified output registers to false.

D0In: d0in **D0Out:** d0out **D1In:** d1in **D1Out:** d1out **D2In:** d2in
Set D registers.

D1In: d1in **D1Out:** d1out
Set D1 registers.

operation: opcode **with:** arg0
operation: opcode **with:** arg0 **with:** arg1
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2 **with:** arg3
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2 **with:** arg3 **with:** arg4
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2 **with:** arg3 **with:** arg4 **with:** arg5
Set up the arguments for a system call. Set the operation to the proper code.

accessing

A0Out
Return the value of the instance variable A0Out.

D0Out
Return the value of the instance variable D0Out.

D1Out
Return the value of the instance variable D1Out.

errors

errorCode
Return an integer which identifies an error condition. This may not be valid if no error has occurred.

execution

displayInvoke
Perform a display system call.

invoke
Perform the system call, return true if it executed without error, return false otherwise. Users of #invoke must check the return value if they expect valid results!

systemInvoke
Make a system call. Return success or failure of that system call. Notify if the primitive failed.

operation type

displayOperation
The desired operation involves the display.

operation
Return the current operation.

operation: opcode
Set the code of the operation.

systemOperation
The desired operation involves the operating system.

portable subtask operations

terminatedSubtaskExitCode
Return a portion of the status returned from the wait system call. This portion represents the value of the argument supplied by the exit system call causing termination. The high order bit of the portion indicates whether the terminated task has made a core dump. The receiver must be an instance representing a wait system call that has been executed. *Subclass responsibility.*

terminatedSubtaskExitInterrupt
Return a portion of the status returned from the wait system call. This portion represents the value of the signal causing termination. The receiver must be an instance representing a wait system call that has been executed. *Subclass responsibility.*

terminatedSubtaskID
Return the ID returned from the wait system call. The receiver must be an instance representing a wait system call that has been executed.

Class Methods

class initialization

initialize
Initialize the class variables used by AimSystemCall subclasses.

environment variables

argCount
Return the number of arguments on the command line used to invoke Smalltalk. *Subclass responsibility.*

originalEnvironment
Return the operating system environment variables in use when Smalltalk was invoked. *Subclass responsibility.*

file names

checkFileName: aFileName **fixErrors:** fixErrors
Check aFileName for validity as a file name. If there are problems (e.g., illegal length or characters) and fixErrors is false, notify an error. If there are problems and fixErrors is true, make the name legal (by omitting illegal characters) and answer the new name. Otherwise, answer the name. Control characters, spaces, rubouts, and all characters with an ASCII value greater than 127 will be considered illegal.

completePathname: aDirectoryName
Answer the complete path name of the string, aDirectoryName. A trailing path name separator is considered part of a directory name even if not explicitly stored.

currentDirectoryPseudonym
Answer the string which represents the name for the current directory.

fileDirectory: aFileDirectory **directoryName:** aFileName
Answer a file directory name derived from the string, aFileName, and the file directory, aFileDirectory.

fullDirectoryName: aDirectoryName
Answer the full path name of the string, aDirectoryName, starting at the directory Disk. A trailing path name separator is considered part of a directory name even if not explicitly stored.

isFileDirectoryName: aFileName
Is aFileName the name of a directory?

isInvisibleFileName: aFileName
Return true if the string, aFileName, is the name of a file which is normally not displayed in a directory listing.

pathNameLevelSeparator
Answer the character which delimits directory names in a file name path.

pathNameLevelSeparatorString
Answer a string containing the character which delimits directory names in a file name path.

separateDirectoryNameAndFileName: aFileName
Split the string, aFileName, into a directory name and a file name within the directory. Return an Array of two elements. Array at: 1 is the directory name, Array at: 2 is the file name.

general inquiries

abnormalTerminationCode
Return the code for abnormal task termination. *Subclass responsibility.*

brokenPipeInterrupt
Return the interrupt number for the broken pipes interrupt. *Subclass responsibility.*

deadChildInterrupt
Return the interrupt number for the terminated child process interrupt. *Subclass responsibility.*

getMachineName
Return the type of machine Smalltalk is running on. *Subclass responsibility.*

nonBlockingWait
Does this class' #wait method return immediately rather than blocking? *Subclass responsibility.*

pack: upperInteger **intoRegisterWith:** lowerInteger
Answer a 32 bit value created by packing the upper and lower Integers together.

priorityInterval
Return the interval of valid priorities in order of descending priority for this task and effective user. *Subclass responsibility.*

returnKeyCode
Answer Smalltalk character value which should be assigned when the return key is pressed. *Subclass responsibility.*

smalltalkInterpreterVersionString
Return a String identifying the Smalltalk interpreter that is currently running.

stringTerminator
Return the Character that must be appended to strings before sending them off to the operating system.

terminateInterrupt
Return the interrupt number for the terminate interrupt. This interrupt can be caught. *Subclass responsibility.*

terminateUnconditionallyInterrupt
Return the interrupt number for an unconditional termination interrupt. This interrupt cannot be caught. *Subclass responsibility.*

textLineDelimiter
Return a String containing the characters that delimit a 'line' in a file.

validPriority: aPriority
Is aPriority a valid priority for this task and user? *Subclass responsibility.*

portable file operations

closeFile: aFileDescriptor
Close the file referred to by aFileDescriptor.

duplicateFd: fileDescriptor
Return a new file descriptor that references the same open file as fileDescriptor. *Subclass responsibility.*

duplicateFd: oldFileDescriptor **with:** newFileDescriptor
Cause newFileDescriptor to reference the same open file as oldFileDescriptor. If newFileDescriptor currently references an open file, that file is first closed. *Subclass responsibility.*

newPipe
Return an instance of Pipe. *Subclass responsibility.*

remove: fileName
Remove the file named fileName.

portable subtask operations

brokenPipesProcessWith: aBlock
Return a process that monitors broken pipes. ABlock is executed after the receipt of each broken pipe signal. *Subclass responsibility.*

execute: program **withArguments:** args **withEnvironment:** environment
Answer an instance of the exec system call. It has not been invoked yet. *Subclass responsibility.*

executeUtility: aCommand **withArguments:** anOrderedCollection
Execute a binary program and return the entire results generated by the program as a string. No mechanism for input to the program is provided. Notify an error if the program cannot be executed or if the program terminates abnormally.

executeUtilityNoCheck: aCommand **withArguments:** anOrderedCollection
Execute a binary program and return the entire results generated by the program as a string. No mechanism for input to the program is provided. Some utilities (rsh, grep) return abnormal termination status after executing successfully — use this method for such programs.

executeUtilityWithErrorMapping: aCommand
withArguments: anOrderedCollection
Execute a binary program and return an array of two strings. The first string contains the entire normal output generated by the program. The second string contains any error message output from the program. No mechanism for input to the executable program is provided. Notify an error if the program cannot be executed or if the program terminates abnormally.

exit: exitParam
Answer an instance of the exit system call. It has not been invoked yet. *Subclass responsibility.*

fork
Answer an instance of the fork system call. It has not been invoked yet. *Subclass responsibility.*

forkShell
Fork an operating system shell. Exit the shell to return to Smalltalk. *Subclass responsibility.*

sendInterrupt: interruptID **to:** taskID
Send the interrupt known by interruptID to the task known by taskID. Return true if the operation was successful, false otherwise. *Subclass responsibility.*

setTaskPriority: priority
Set the priority of Smalltalk to the value priority. *Subclass responsibility.*

startSubtask: executeCall **withBlock:** childBlock
Fork a copy of Smalltalk. In the child copy, execute childBlock and invoke executeCall, which must be an instantiated 'exec' system call. *Subclass responsibility.*

terminate: taskID
Using an interrupt, attempt to terminate the task known by taskID. This termination is requested in a manner which can be intercepted. *Subclass responsibility.*

terminatedSubtasksProcessWith: aBlock
Return a Process that monitors spawned child tasks. ABlock is executed after the termination of each child task. The dead child signal is automatically reset by the operating system. *Subclass responsibility.*

terminateUnconditionally: taskID
Terminate this task unconditionally. *Subclass responsibility.*

wait
Answer an instance of the wait system call. It has not been invoked yet. *Subclass responsibility.*

system-display operations

blackOnWhite
Normal video.

cursorOff
Turn off the cursor.

cursorOn
Turn on the cursor.

disableCursorPanning
Disables panning when the cursor reaches any viewport boundary.

disableJoydiskPanning
Disables panning with the joydisk.

enableCursorPanning
Enables panning when the cursor reaches any viewport boundary.

enableJoydiskPanning
Enables panning with the joydisk.

getDisplayState: buff
Return the 36 word status report of the display system.

getMouseBounds
Get the limits of mouse movement as a rectangle — upper left in D0, lower right in D1.

getViewport
Return the upper left position of the viewport within the display bitmap.

setMouseBounds: upperXY **lowerRight:** lowerXY
Set the limits of mouse movement to a rectangle — upper left in D1, lower right in D2.

setViewport: xyCoord
Set the upper left position of the viewport in the display bitmap.

timeOutOff
Disables automatic screen saver.

timeOutOn
Enables automatic screen saver.

turnDisplayOff
Sets the display to be blanked.

turnDisplayOn
Sets the display to be visible.

whiteOnBlack
Inverse video.

system-environment

configurationAddress
Get the address of the configuration block of the machine running this Smalltalk.

copyrightAddress
Get the address of the copyright string for this Smalltalk interpreter.

interpreterVersionAddress
Get the address of the string identifying the version of this Smalltalk interpreter.

invocationArgumentsAddress
Get the address of the argument array for this invocation of Smalltalk.

invocationEnvironmentAddress
Get the address of the environment array for this invocation of Smalltalk.

machineIDAddress
Get the address of a string that identifies the operating system and machine upon which Smalltalk is running.

timeCorrectionDifference
Get a SmallInteger representing the number of seconds difference between UTek time (GMT) and local time.

system-event operations

clearAlarm
Clear the alarm used by the graphics.

eventsDisable
Turn off events.

eventsEnable
Turn on events. A side effect of this call is to set keyboard code to zero.

eventSignalOn
Produce event signals.

getAlarmTime
Get the millisecond timer.

setAlarmTime: time
Set the millisecond timer to the desired time.

setKeyboardCode: aCode
Set the keyboard code — 0 sets keyboard to output event codes and 1 sets the keyboard to output ANSI terminal code. Event codes are what Smalltalk expects.

terminalOff
Turn off the terminal emulator.

terminalOn
Turn on the terminal emulator.

Rationale

AimSystemCall is an abstract class that implements protocol related to the display subsystem of Tektronix 4000 series bitmapped workstations. These display operations are implemented at this level to be inherited by concrete subclasses for various operating systems. Other system call protocol that concrete subclasses have in common has been implemented at this level. Instance variables are defined for data passed in registers when making a system call. A class variable is defined for the string terminator that subclass operating systems have in common.

Discussion

The protocol here is a template for subclasses. There is no occasion upon which you should send a message to this class. To make a system call, create an instance of a subclass. Read about the system call class for your operating system in this manual. It will implement the protocol found here, plus additional protocol which is operating system dependent.

Class Protocol

Class initialization contains one method which sets the value of the class variable. It calls the abstract superclass method of the same name, which returns self. The major work of initializing is done in a subclass method.

Environment variables methods are implemented by a subclass.

File names methods answer information about file and directory names, including the complete path from root, the path relative to **Disk**, the character which separates levels of the path, the backup file suffix, and concatenates or separates file and directory names. These methods are generally used for parsing.

General inquiries methods are implemented by a subclass, except for **stringTerminator**, which returns the character the operating systems expect at the end of strings, **pack:intoRegisterWith:**, which packs two integers into a 32 bit word, and **smalltalkInterpreterVersionString**, which returns the version number of the interpreter in a **String**.

Portable file operations methods are implemented by a subclass, except for **closeFile:**, which closes a file, and **remove:**, which removes a file from its directory. Those two operations are implemented here because subclasses use the same selector to achieve the result of each portable message.

Portable subtask operations are operations that Smalltalk requires from the operating system in order to run a **Subtask**, a spawned child process from the operating system's point of view. **AimSystemCall** implements the "executeUtility" methods which start a subtask. The other methods in this message category are implemented by a subclass.

System-display operations are methods which enable or disable display attributes. They are implemented here because they are applicable for all immediate subclasses.

System-environment methods return the address of environment variables at the time Smalltalk was invoked, the address of command line arguments, and information about the software, including the copyright address, the address of the interpreter version, and the adjustment amount between UTek and Smalltalk time. An address returned by one of these methods can be converted to a **String** by sending the address as the argument to the **String** *instance creation* message **fromCString:**.

System-event operations are methods which enable or disable event attributes. Events are user-interface occurrences, such as a key-press, mouse movement, and joydisk movement. The built-in alarm (timer) and terminal emulator are also included in the category of events. Events are implemented here because they are applicable for all immediate subclasses.

Instance Protocol

Initialize-release has three methods which set the values of the register instance variables plus several methods, overridden by subclasses, which set up the operation and arguments to a system call.

Accessing methods return the values of the "out" instance variables.

Errors has one method that returns the value of **errno.**

Execution has two methods, **displayInvoke** and **invoke**, which call the proper primitive to communicate with the operating system, and **systemInvoke** which calls a subclass method to make a primitive call.

Operation type has methods to set the **operationType** instance variable and a method that sets and one that returns the value of **operation**.

Portable subtask operations are operations which an instance of a subclass would perform for a **Subtask**, such as answer the exit code, the interrupt ID, or the task ID of the **Subtask**. The methods are implemented by a subclass, except **terminatedSubtaskID**, which returns the value of **D0Out**.

Related Classes

Subclasses:
 UniflexSystemCall (UniFLEX only)
 UTekSystemCall (UTek only)

You might also want to look at the superclass, **AbstractSystemCall.**

```
ExternalBinaryData variableByteSubclass: #DisplayState

    instanceVariableNames:      ''
    classVariableNames:         'BlackOnWhiteBitPosition CursorDataIndex
                                CursorLinkedBitPosition CursorOffsetXDataIndex
                                CursorOffsetXYDataIndex
                                CursorOffsetYDataIndex CursorOnBitPosition
                                CursorPanningBitPosition DisplayOnBitPosition
                                EventsBitPosition HeightDataIndex
                                JoyPanningBitPosition KeyboardCodeDataIndex
                                KeyCapsLockLEDBitPosition
                                LineIncrementDataIndex
                                MouseBoundLrDataIndex
                                MouseBoundUlDataIndex
                                ScreenSaverBitPosition StateBitsDataIndex
                                TerminalEmulatorBitPosition TotalSizeDataIndex
                                ViewportDataIndex ViewPortHeightDataIndex
                                ViewPortWidthDataIndex WidthDataIndex '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

DisplayState captures all of the information about the state of the display in a single class. This class is represented in the operating system as the C structure below. **DisplayState** provides creation and accessing protocol for the structure.

```
struct displaystate {
    int     disp_stateBits;         /* bit definitions as below */
    int     disp_viewport;          /* upper left corner point of viewport */
    int     disp_mouseBound_ul;     /* upper left corner point of mouse
                                       bounds */
    int     disp_mouseBound_lr;     /* lower right corner point of mouse
                                       bounds */
    short   disp_cursor[16];        /* the cursor image */
    char    disp_keyboardCode;      /* current keyboard encoding:
                                       0=events, 1=ansi */
    char    disp_reserved1;         /* reserved for future use */
    short   disp_lineIncrement;     /* number of bytes between scan lines */
    short   disp_width;             /* width of virtual display bitmap */
```

```
        short   disp_height;            /* height of virtual display bitmap */
        short   disp_viewPortWidth;     /* width of viewport */
        short   disp_viewPortHeight;    /* height of viewport */
        short   disp_cursorOffsetX;     /* X graphic cursor offset */
        short   disp_cursorOffsetY;     /* Y graphic cursor offset */
        int     disp_reserved[2];       /* reserved for future use */
    }
```

stateBits Definition

name	bit position *	comment
displayOn	1	1 means on / 0 means off
screenSaver	2	1 means screen saver on / 0 means off
blackOnWhite	3	1 means black on white / 0 means white on black
terminalEmulator	4	1 means emulator output enabled / 0 disabled
keyCapsLockLED	5	1 means the Caps Lock LED is illuminated
cursorOn	9	1 means graphics cursor is enabled
cursorLinked	10	1 means mouse is linked to cursor
cursorPanning	11	1 means cursor movement can cause panning
joyPanning	12	1 means joydisk causes viewport panning
events	17	1 means events mechanism is turned on

* Smalltalk bit positions are counted from position 1, not 0 as in the C structure statebits field.

The structure is documented under *getDisplayState* in the system call section of the manual that documents platform-specific extensions for your operating system.

Class Variables

BlackOnWhiteBitPosition

CursorDataIndex

CursorLinkedBitPosition

CursorOffsetXDataIndex

CursorOffsetXYDataIndex

CursorOffsetYDataIndex

CursorOnBitPosition

CursorPanningBitPosition

DisplayOnBitPosition

EventsBitPosition

HeightDataIndex

JoyPanningBitPosition

KeyboardCodeDataIndex

KeyCapsLockLEDBitPosition

LineIncrementDataIndex

MouseBoundLrDataIndex

MouseBoundUlDataIndex

ScreenSaverBitPosition

StateBitsDataIndex

TerminalEmulatorBitPosition

TotalSizeDataIndex

ViewportDataIndex

ViewPortHeightDataIndex

ViewPortWidthDataIndex

WidthDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **StateBitsDataIndex** holds the offset of the "disp_stateBits" field. For this structure, each bit position in the "disp_stateBits" field has its own class variable.

Instance Methods

accessing

cursorOffsetX
Return the value of the structure field named cursorOffsetX.

cursorOffsetY
Return the value of the structure field named cursorOffsetY.

height
Return the value of the structure field named height.

keyboardCode
Return the value of the structure field named keyboardCode.

lineIncrement
Return the value of the structure field named lineIncrement.

mouseBoundLr
Return the value of the structure field named mouseBoundLr.

mouseBoundLrX
Return the X coordinate of the value of the structure field named mouseBoundLr.

mouseBoundLrY
Return the Y coordinate of the value of the structure field named mouseBoundLr.

mouseBoundUl
Return the value of the structure field named mouseBoundUl.

mouseBoundUlX
Return the X coordinate of the value of the structure field named mouseBoundUl.

mouseBoundUlY
Return the Y coordinate of the value of the structure field named mouseBoundUl.

stateBits
Return the value of the structure field named stateBits.

viewport
Return the value of the structure field named viewport.

viewPortHeight
Return the value of the structure field named viewPortHeight.

viewPortWidth
Return the value of the structure field named viewPortWidth.

viewportX
Return the X coordinate value of the structure field named viewport.

viewportY
Return the Y coordinate value of the structure field named viewport.

width
Return the value of the structure field named width.

accessing-status

blackOnWhite
Answer true if the blackOnWhite state bit is set.

cursorLinked
Answer true if the cursorLinked state bit is set.

cursorOn
Answer true if the cursorOn state bit is set.

cursorPanning
Answer true if the cursorPanning state bit is set.

displayOn
Answer true if the displayOn state bit is set.

events
Answer true if the events state bit is set.

joyPanning
Answer true if the joyPanning state bit is set.

keyCapsLockLED
Answer true if the keyCapsLockLED state bit is set.

screenSaver
Answer true if the screenSaver state bit is set.

terminalEmulator
Answer true if the terminalEmulator state bit is set.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize

Assign offset values to the class variables and define the size of the structure.

Rationale

The display state structure stores in one place all important attributes that affect the current environment of the display. The structure is used in support of the following system call:

getDisplayState

Discussion

DisplayState does not provide protocol to modify the C structure in the operating system. Its purpose is to provide access to the information in the structure.

AimSystemCall provides protocol for changing the state of the display. Under *system-display operations* you will find methods to enable and disable display attributes, for example, inverse/normal video, cursor panning, and joydisk panning.

See the manual that documents platform-specific extensions for your operating system for more information about display attributes and the fields in the displaystate structure.

Related Classes

AimSystemCall implements the system call *getDisplayState* which uses the displaystate structure.

```
ExternalData variableByteSubclass: #ExternalBinaryData

    instanceVariableNames:      ''
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

ExternalBinaryData is an abstract class for non-Smalltalk data structures that do not have imbedded pointers. The concrete classes representing non-Smalltalk data structures are used to pass information between Smalltalk and the operating system, for example, when making system calls. The indexable fields of instances of subclasses represent the same bytes as the named fields of the external language's data structure. This is explained in the "Discussion" section.

Subclasses implement creation and accessing methods for specific non-Smalltalk data types such as displaystate, a C structure, and wait, a C union.

Instance Methods

accessing

dataArea
Answer the data portion of the receiver.

replaceFrom: start **to:** stop **with:** replacement **startingAt:** repStart
This destructively replaces elements from start to stop in the receiver, starting at index repStart in the Collection replacement. Answer the receiver.

conversion

asStringFrom: start **to:** stop
Create a string from the receiver's bytes between start and stop, inclusive.

```
ExternalBinaryData class

    instanceVariableNames:      'sizeInBytes '
```

ExternalBinaryData class — Instance Variables

sizeInBytes <Integer>

Number of bytes required to represent the data area of an instance.

Class Methods

accessing

numberOfPointers

Return the number of pointers imbedded in the structure represented by an instance of the receiver.

pointersSize

Answer the pointers size of any instance.

sizeof

Answer the number of bytes in the data section of any instance.

instance creation

new

Return a new instance of the receiver.

Rationale

Since all external data classes either do or do not contain pointers, the **ExternalData** hierarchy splits at this level into **ExternalBinaryData** and **ExternalPointerData**. **ExternalBinaryData** is an abstract class which implements protocol common to all of its concrete subclasses which, by definition, do *not* contain pointers. Inherited protocol designated as subclass responsibility is implemented in this class. All subclasses of the **ExternalBinaryData** branch of the external data hierarchy have a class instance variable, **sizeInBytes**, defined here as the number of bytes in the data area of all instances of a subclass.

Discussion

Naming Conventions for Subclasses' Class Variables

Most of the time (exceptions are explained below), there is one class variable for each named field in the external data structure represented by a Smalltalk class. Each external data structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. The bytes of a named field in an external structure reside in the byte array starting at the offset indicated by the class variable for the field. For example, in the following figure, the bytes of a field called "foo" would begin at position FooDataIndex in the data area (byte array) of the

Phoney object representing the structure with "foo" in it. See Figure 1 for a simple example.

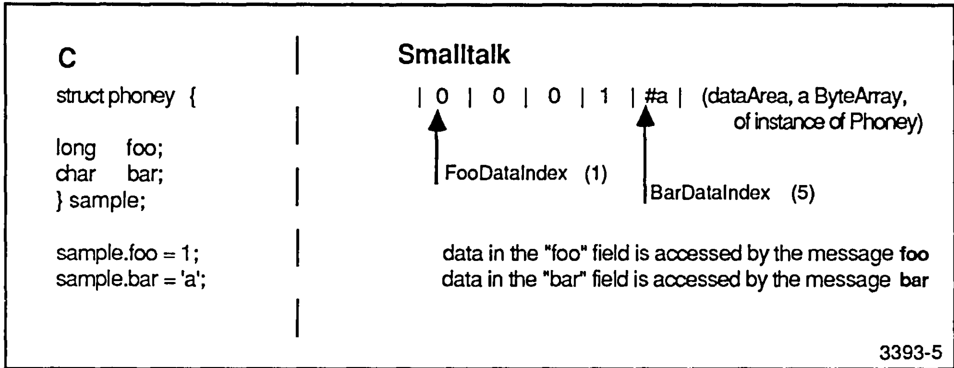


Figure 1

An example will help to illustrate the class variables naming convention. The following is the definition of the C structure which the **Rusage** class represents.

```
struct rusage {
    struct timeval  ru_utime;     /* user time used */
    struct timeval  ru_stime;     /* system time used */
    long            ru_maxrss;
    long            ru_ixrss;     /* integral shared memory size */
    long            ru_idrss;     /* integral unshared data size */
    long            ru_isrss;     /* integral unshared stack size */
    long            ru_minflt;    /* page reclaims */
    long            ru_majflt;    /* page faults */
    long            ru_nswap;     /* swaps */
    long            ru_inblock;   /* block input operations */
    long            ru_oublock;   /* block output operations */
    long            ru_msgsnd;    /* messages sent */
    long            ru_msgrcv;    /* messages received */
    long            ru_nsignals;  /* signals received */
    long            ru_nvcsw;     /* voluntary context switches */
    long            ru_nivcsw;    /* involuntary */
}
```

The first two fields of struct rusage are of type struct timeval, defined as follows.

```
struct timeval {
    long   tv_sec;     /* seconds */
    long   tv_usec;    /* microseconds */
}
```

There are fourteen class variables for the **Rusage** fields with simple data types (all the fields of type long). In addition to the fourteen class variables for the simple fields of struct rusage, **Rusage** has class variables for the fields whose data type is a structure — rusage fields ru_utime and ru_stime are of type struct timeval. The additional variables are named **StimeSecDataIndex**, **StimeUsecDataIndex**, **UtimeSecDataIndex**, and **UtimeUsecDataIndex**. These names are derived from the field name in struct rusage and the name of the field in struct timeval. Field names are stripped of their prefix, as usual, to arrive at the "structure within a structure" class variable names.

If an external data structure has fields which are not presently used, usually indicated by "reserved" in the field comment, the Smalltalk class for the structure does *not* have class variables for the "reserved" fields.

Naming Conventions for Subclasses' Protocol

Each concrete subclass of **ExternalBinaryData** has protocol for accessing the data of the structure. Message selectors are the field name, stripped of its prefix. For example, sending the message **inblock** to an instance of **Rusage** will return the value of the ru_inblock field. The message **sec:**, sent to an instance of **Timeval**, will set the value of the tv_sec field to the specified argument. Depending upon whether the structure is exclusively filled in by the operating system, or whether Smalltalk will send data to the operating system in the structure, accessing methods are provided to access or to access *and set* the values of fields. In the case of **Rusage**, protocol is provided to only access the field data, not to set the fields, since the purpose of the structure is to obtain data from the operating system. Since a **Timeval** is used to send and receive data, protocol is provided to access and set the values of fields.

When the value of a field is dependent upon the value of another field, no separate method is provided to set the value of the dependent field. The method which sets the value of the field it depends upon sets the value of the dependent also. This pattern of accessing protocol for field values which are dependent upon other fields has been followed in the concrete subclasses of **FixedSizeExternalPointerData** and **ExternalBinaryData.**

When a structure field is another structure, separate accessing messages are provided for the fields in the imbedded structure. For example, the **Rusage** method **stimeSec** returns the value of the tv_sec field of the timeval structure addressed by the struct rusage ru_stime field.

Protocol to set the value of a field which is an imbedded structure takes an instance of the class representing the imbedded structure as an argument. For example, the **Itimerval** method **interval:** takes an instance of **Timeval** as an argument and sets the value of the it_interval field, which is of type struct timeval.

ExternalBinaryData Class Protocol

Accessing methods implement inherited protocol to return the **numberOfPointers** (0 for all subclasses) and the **sizeOf** all instances of a subclass, the **sizeInBytes** class instance variable. The **pointersSize** method exists to parallel the **ExternalPointerData** method of that name; it returns 0 since no subclasses contain pointers.

Instance creation defines the method **new** to return an instance with the correct number of indexable variables. As an abstract class, you would not send the message **new** to **ExternalBinaryData** to create an instance of this class. The method is implemented here for use in subclasses. *Instance creation* methods in concrete subclasses return an instance of the class with the fields filled, either with data obtained via a system call or data supplied as arguments to the instance creation message. There is no pattern for subclasses' *instance creation* protocol — it varies according to the common uses of individual structures.

ExternalBinaryData Instance Protocol

Accessing has one method, **dataArea**, which returns the receiver, since all instances of subclasses are strictly data (no pointers). Subclasses of the other branch of external data, **ExternalPointerData**, have an instance variable, **dataArea**, which is returned when the message **dataArea** is sent to an instance of a subclass. The inherited method **replaceFrom:to:with:startingAt:** is reimplemented; it invokes a primitive to perform a memory to memory data copy.

Conversion has one method which converts a specified range of the receiver from bytes to a **String** and returns the **String**. The purpose of this method is to convert a string as represented in an external language (for example, an array of chars in C) to a Smalltalk **String**.

Adding Classes to this Hierarchy

If you are adding a subclass of **ExternalBinaryData** and the conventions just described are unclear to you, the best idea is to find a structure class similar to the structure you are adding and model your class on the similar one. Your new class' protocol must include a class initialization method which initializes the class variables and **sizeInBytes**. Remember to execute the **initialize** method for the new class.

Related Classes

Subclasses:

DisplayState
Inaddr
IntegerPointer
Itimerval
Ltchars
Rlimit
Rusage
Sgttyb
SockaddrIn
SockaddrUn
Stat
Tchars
Timeval
Timezone
Utsname
Wait

You might also want to look at **ExternalData**, the superclass. If you are adding a class for a structure with imbedded pointers, read about **ExternalPointerData** — your new class should be a subclass of that class or its subclass, **FixedSizeExternalPointerData**.

Object subclass: #ExternalData

instanceVariableNames:	''
classVariableNames:	'Char Int Long Pointer Short '
poolDictionaries:	''
category:	'OS-Parameters'

Summary

ExternalData is an abstract class for defining data structures suitable for communicating with non-Smalltalk languages. Translation must occur to change Smalltalk objects to the data structure of another language, and vice versa. This class reimplements some **ByteArray** protocol.

Naming Conventions for Subclasses and Subclass Protocol

A class is created for each external language structure. A structure with imbedded machine pointers is made a subclass of **ExternalPointerData**. A structure without imbedded machine pointers is made a subclass of **ExternalBinaryData**. The class name is the same as the structure name. For example, the class **SockaddrIn** provides creation and accessing protocol for a C structure of that name. Fields are accessed by the names given in the structure, stripped of the prefix. For example, in **SockaddrIn**, the "sin_family" field is accessed with the message **family**. Offsets into the structure are provided as class variables.

Class Variables

Char <Integer>
Number of bytes for a C char type.

Int <Integer>
Number of bytes for a C int type.

Long <Integer>
Number of bytes for a C long type.

Pointer <Integer>
Number of bytes for a C pointer type.

Short <Integer>
Number of bytes for a C short type.

Instance Methods

accessing

at: index
Answer the value of the indexable field, index, of the receiver's dataArea.

at: index **put:** anObject
Store the value, anObject, in the indexable field, index, of the receiver's dataArea.

dataArea
Answer the data portion of the receiver. *Subclass responsibility.*

doubleWordAt: index **put:** value
Set the value of the double word (4 bytes) starting at byte index.

numberOfPointers
Return the number of pointers imbedded in the structure represented by the receiver.

replaceFrom: start **to:** stop **with:** replacement **startingAt:** repStart
This destructively replaces elements from start to stop in the receiver, starting at index repStart in the Collection replacement. Answer the receiver. No range checks are performed.

signedIntegerDoubleWordAt: index
Answer the value of the double word (4 bytes) starting at byte index. Treat the value as a 2's complement signed integer.

signedIntegerWordAt: index
Answer the value of the word (2 bytes) starting at byte index. Treat the value as a 2's complement signed integer.

sizeof
Answer the number of bytes in the data section of the receiver.

unsignedIntegerDoubleWordAt: index
Answer the value of the double word (4 bytes) starting at byte index. Treat the value as an unsigned integer.

unsignedIntegerWordAt: index
Answer the value of the word (2 bytes) starting at byte index. Treat the value as an unsigned integer.

wordAt: index **put:** value
Set the value of the word (2 bytes) starting at byte index.

Class Methods

class initialization

initialize
Initialize the class variables.

accessing

numberOfPointers
Return the number of pointers imbedded in the structure represented by the receiver. *Subclass responsibility.*

sizeof
Answer the number of bytes in the data section of any instance. *Subclass responsibility.*

Rationale

ExternalData is an abstract class which implements protocol common to all of its subclasses. This class and its subclasses were created to provide a mechanism for Smalltalk to exchange data with the operating system when making system calls.

Discussion

As an abstract class, there is no instance creation protocol in **ExternalData**. The instance and class protocol includes message selectors that will be sent to subclasses, not to this class. If you are adding a class that represents an external language structure, you should look at the abstract subclasses of **ExternalData** and make your new class a subclass of one of them, not a subclass of this class.

Class Protocol

Class initialization has one method which sets the values of the class variables.

Accessing methods return the number of pointers in the structure, and the number of bytes in the **dataArea** of the structure — implementation of these functions is left to a subclass.

Instance Protocol

Since the data of an external structure are represented in a Smalltalk object as a **ByteArray**, this class reimplements some **ByteArray** accessing protocol.

Accessing methods return or set the values of indexable fields in the **dataArea** of the receiver. There are different accessing methods for the different sizes of data — one byte, two bytes, or four bytes — and for signed or unsigned integers. One method, **replaceFrom:to:With:startingAt:**, replaces a chunk of data in the receiver. The work of two methods, **dataArea** and **sizeof**, is ultimately done by a subclass.

Related Classes

Subclasses:

- **ExternalBinaryData**
- **ExternalPointerData**

```
ExternalData subclass: #ExternalPointerData
    instanceVariableNames:      'dataArea pointers '
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

ExternalPointerData is an abstract class for a non-Smalltalk data structure containing machine pointers. It holds the binary data for the non-Smalltalk data structure and an array of Smalltalk objects whose machine addresses will be inserted into the binary data. The addresses are inserted by the interpreter when certain primitives are invoked.

Instance Variables

dataArea <ByteArray>
Binary data of a non-Smalltalk data structure.

pointers <Array>
Pairs offsets into the binary data portion (dataArea) with objects.

Instance Methods

accessing

dataArea
Return the dataArea of the receiver.

dataArea: aByteArray
Update the dataArea of the receiver with aByteArray.

dataArea: aByteArray **pointers:** anArray
Update the dataArea of the receiver with aByteArray and the pointers with anArray.

numberOfPointers
Return the number of pointers imbedded in the structure represented by

the receiver.

pointers
Return the pointers of the receiver.

pointers: anArray
Update the pointers of the receiver.

sizeof
Return the size of the data area in bytes.

copying

copy
Return a copy of the receiver. Insure that pointers of the copy reference the same objects as the pointers of the receiver.

Rationale

Since all external data classes either do or do not contain pointers, the **ExternalData** hierarchy splits at this level into **ExternalBinaryData** and **ExternalPointerData**. **ExternalPointerData** is an abstract class which implements protocol common to all of its concrete subclasses which, by definition, contain one or more pointers. Inherited protocol designated as subclass responsibility is implemented in this class.

Discussion

Naming Conventions for Subclasses' Class Variables

The name of a class variable is constructed from a field name, stripped of its prefix, with one of the following strings appended, as appropriate: 'DataIndex' or 'PointerIndex'. Fields without pointers have one class variable, <field-name>DataIndex. For fields with a data type having imbedded machine pointers, two class variables are created:

- one for the offset of the binary data, and
- one for the index of the pointer to that field, with respect to other pointers in the structure.

The <field-name>DataIndex indicates the index of the first byte of the data in the **dataArea**. The <field-name>PointerIndex is used in accessing protocol to access or set the value of a pointer field in **pointers**.

An example will help to illustrate the naming convention for class variables. The following is the definition of the C structure which the **Msghdr** class represents.

```
struct msghdr {
    caddr_t       msg_name;            /* optional address */
    int           msg_namelen;         /* size of address */
    struct iovec  *msg_iov;            /* scatter/gather array */
    int           msg_iovlen;          /* # elements in msg_iov */
    caddr_t       msg_accrights;       /* access rights sent/received */
    int           msg_accrightslen;
}
```

The third field, *msg_iov, is a pointer to a struct iovec, which is defined as follows.

```
struct iovec {
    caddr_t   iov_base;
    int       iov_len;
}
```

There are six fields in struct msghdr. The first, third and fifth fields are pointer types (caddr_t is defined as a pointer to a char). The class variables for the pointer fields are shown in the following table.

Pointer Fields' Class Variables		
field name	class variables	contents
msg_name	**NameDataIndex**	offset (1) of the first byte of *msg_name* data in **dataArea**
	NamePointerIndex	index (1) of this pointer with respect to other pointers in the structure
*msg_iov	**IovDataIndex**	offset (9) of the first byte of **msg_iov* data in **dataArea**
	IovPointerIndex	index (2) of this pointer with respect to other pointers in the structure
msg_accrights	**AccrightsDataIndex**	offset (17) of the first byte of *msg_accrights* data in **dataArea**

AccrightsPointerIndex	index (3) of this pointer with respect to other pointers in the structure

Msghdr has three other class variables for the three fields which are not pointers.

Other Fields' Class Variables		
field name	class variable	contents (offset)
msg_namelen	**NamelenDataIndex**	5
msg_iovlen	**IovlenDataIndex**	13
msg_accrightslen	**AccrightslenDataIndex**	21

If an external data structure has fields which are not presently used, usually indicated by "reserved" in the field comment, the Smalltalk class for the structure does *not* have class variables for the "reserved" fields.

dataArea and pointers Instance Variables

This section describes the two instance variables, **dataArea** and **pointers**, provides an illustration of their relationship, and gives details of the elements of the value-offset pairs in **pointers**.

The **dataArea** instance variable is a **ByteArray** of the data in the structure. The index of the first byte of each structure field in the **dataArea** is indicated by the <field-name>DataIndex class variable for the field. A pointer field's "DataIndex" class variable is also the second element of the field's value-offset pair in the **pointers** array.

The pointer array, **pointers**, is an instance of **Array** which stores pairs of elements (i.e., "anArray at: 1:" together with "anArray at: 2" make up a "pair"). For each pointer in a structure, there is one pair (two elements) in **pointers**. For example, since a **Msghdr** contains three pointers, its **pointers** array has three pairs — six elements.

The figure below depicts an instance of **Msghdr** to show the value of its instance variables. The code preceding the figure is an example that creates the objects needed for the figure.

```
anIovecArray ← StructureArray new: 2 class: Iovec.
anIovecArray at: 1 put: (Iovec base: 'first').
anIovecArray at: 2 put: (Iovec base: 'second').
```

msghdr ← Msghdr name: 'foo' iov: anIovecArray accrights: 'rights'.

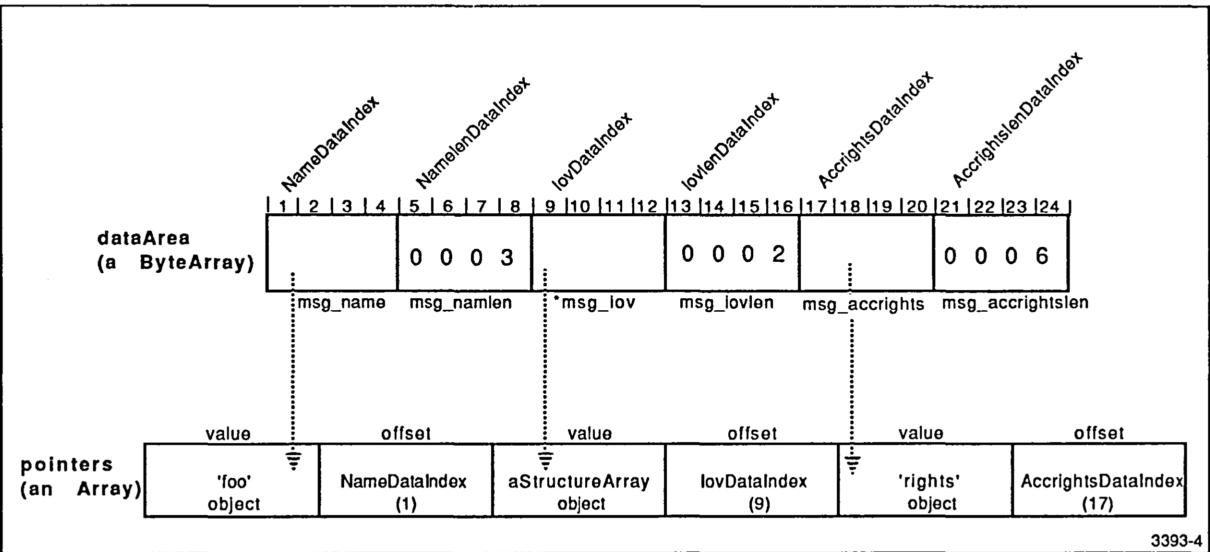


Figure 1. An instance of Msghsr

The spaces reserved for addresses in the **dataArea** are filled by the system/display call primitive. The dotted lines point to the Smalltalk objects whose addresses are in the **dataArea.**

In the figure, the **dataArea** has space for three machine addresses which will be filled by a system call or display primitive. The dotted lines indicate that the addresses filled in by the primitive will refer to objects in the **pointers** array.

Value

The first element of a pair (the odd indexed element) contains a Smalltalk object which is conceptually referenced by the field of the non-Smalltalk structure. This element is called the *value*. If the function of the system call is to return data from the external language structure to Smalltalk, then the appropriate *value* locations in **pointers** will be changed after the system call.

Offset

The second element of a pair (the even indexed element) is called the *offset*. It contains the index into the **dataArea**, a **ByteArray**, where the first byte of the pointer (i.e., an address) is stored. Indices in the **dataArea** are counted from 1. The offset identifies the position where a reference to a Smalltalk object is considered to *virtually* exist. The space reserved for addresses in **dataArea** is filled in by a display or system call primitive; however, after the call is completed the addresses may no longer be valid for the Smalltalk objects they referenced when the call was invoked.

Primitive Action on dataArea

If the offset is nil, indicating an unused map pair, the associated value is ignored. This might be useful for representing ´union´ types in which some fields must be empty. If the value is the object nil, the primitive will replace the associated field in the **dataArea** with the binary value 0. If the value is a **SmallInteger** or a 4-byte **LargePositiveInteger** or **LargeNegativeInteger**, the primitive will replace the associated field in the **dataArea** with the machine representation of the integer value. If the value is an **ExternalBinaryData** object (or a variableByte or variableWord class without instance variables), then the field is replaced with the machine address of the data part of the object. Otherwise, the value is assumed to be an **ExternalPointerData** object; the associated field in the **dataArea** is replaced with the address of the data part of the **ExternalPointerData** object and above rules are recursively applied. When the recursion is completed, all pointers in the **dataArea** will contain valid addresses and the pointers in the **dataArea** of each *referenced object* will also contain valid addresses. This allows a structure to contain pointers to other structures.

The following table summarizes the possible meanings of the **pointers** array.

Offset	Value	Primitive Action on **dataArea**
nil	*anything*	value is ignored
x	nil	at x store the machine integer whose value is 0
x	SmallInteger	at x store machine integer
x	4-byte LargePositiveInteger	at x store machine integer
x	4-byte LargeNegativeInteger	at x store machine integer
x	ExternalBinaryData	at x store machine address of data portion
x	ExternalPointerData	at x store machine address of data portion and recurse

(x is an integer)

Naming Conventions for Subclasses' Protocol

Each concrete subclass of **ExternalPointerData** has protocol for accessing the data of the structure. Message selectors are the field name, stripped of its prefix. For example, sending the message **accrights** to an instance of **Msghdr** will return the value of the msg_accrights field. The message **accrights:**, sent to an instance of **Msghdr**, will set the value of the msg_accrights field to the specified argument. Since the value of the msg_accrightslen field is dependent upon the length of the string in msg_accrights, there is no method provided to set the value of msg_accrightslen. The **accrights:** method sets the value of both fields. This pattern of accessing protocol for field values which are dependent upon other fields has been followed in the concrete subclasses of **FixedSizeExternalPointerData** and **ExternalBinaryData.**

Depending upon whether the structure is exclusively filled in by the operating system, or whether Smalltalk will send data to the operating system in the structure, accessing methods are provided to access or to access *and set* the values of fields.

Protocol to set the value of a field which is an imbedded structure takes an instance of the class representing the imbedded structure as an argument. At present there are no subclasses of **FixedSizeExternalPointerData** which represent a structure with an imbedded structure.

ExternalPointerData Instance Protocol

This protocol is present to be inherited by concrete subclasses. Since this is an abstract class, there is no instance creation protocol and no messages should be sent to this class.

Accessing has one method, **dataArea**, which returns the **dataArea** instance variable, and another method to set the **dataArea**. There are two methods to access and set the value of the **pointers** instance variable. Another method, **dataArea:pointers:**,

sets the value of both instance variables. The **numberOfPointers** method returns the number of pointers in the structure, and the **sizeOf** method answers the size of the **dataArea** in bytes.

Copying has one method which returns a copy of the receiver.

Related Classes

Subclasses:

FixedSizeExternalPointerData
PointerArray
StructureArray

You might also want to look at the superclass, **ExternalData**, and the parallel abstract class for structures which do not contain pointers, **ExternalBinaryData**.

```
FileStream subclass: #FileDirectory

    instanceVariableNames:      ''
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Streams'
```

Summary

Instances of **FileDirectory** are a special kind of **FileStream** that represent directories. The instance variable **name** identifies the directory instances refer to. Directories can be viewed as a collection of files — enumerating protocol is provided. Instances of **FileDirectory** can be found in dictionaries, or another **FileDirectory**, though often this is implicit.

Inherited Instance Variables

name <String>
: This inherited instance variable contains a relative or absolute path specifying this directory. Relative paths are relative to Disk.

directory <nil>
: This inherited instance variable is not used.

Instance Methods

accessing

completePathname
: Answer the complete path name of the receiver, starting with the root directory. A trailing path name separator is considered part of a directory name even if not explicitly stored.

contents
: Answer the names of all files in this directory.

directoryName
: Answer the name of the receiver.

fullName
: Answer the full path name of the receiver, relative to Disk if appropriate. A trailing path name separator is considered part of a directory name even if

not explicitly stored.

versionNumbers
Answer true if version numbers are supported.

adding

addKey: aFileName
Create a new file whose name is aFileName. The method newFile: produces an error if the file already exists.

enumerating

do: aBlock
Sequence over all possible files (or directories) in the receiver, evaluating aBlock for each one.

filesMatching: patternString
Answer an Array of the names of files (or directories) that match patternString.

namesDo: aBlock
Sequence over all possible file (or directory) names in the receiver, evaluating aBlock for each one. A collection of file names is created so that the operating system will not dynamically increase the directory size when backup files are created.

file accessing

checkName: aFileName **fixErrors:** fixErrorsBoolean
Check aFileName for validity as a file in this directory.

directoryNamed: aString
Answer an instance of myself whose name is aString. Answer Disk if aString matches Disk.

file: aFileName
Answer a FileStream on an old or new file whose name is aFileName.

fileClass
Answer the proper class whose instances represent files in the receiver.

isLegalFileName: aString
Answer whether aString is a legal file name.

isLegalOldFileName: aString
Answer whether aString is a legal file name and if the file exists in the receiver.

newDirectory: aDirectoryName
Answer a FileDirectory on a new directory whose name is aDirectoryName; notify if the argument is not a new file name.

newFile: aFileName
Answer a FileStream on a new file whose name is aFileName; notify if the argument is not a new file name.

oldFile: aFileName
Answer a FileStream on an old file whose name is aFileName; notify if the argument is not an old name.

oldWriteOnlyFile: aFileName
Answer a FileStream on an old file (write only) whose name is aFileName; notify if the argument is not an old name.

rename: aFile **newName:** newName
Rename the file, aFile, to have the name newName; notify if a file by the name, newName, already exists.

file copying

append: aFileName1 **to:** aFileName2
Append the contents of a file whose name is aFileName1 to the end of a file whose name is aFileName2.

copy: aFileName1 **to:** aFileName2
Copy the contents of a file whose name is aFileName1 to a file whose name is aFileName2.

removing

removeKey: aFileName
Remove the file whose name is aFileName; notify if not found.

removeKey: aFileName **ifAbsent:** absentBlock
Remove the file whose name is aFileName; answer the result of evaluating absentBlock if not found.

testing

includesKey: aFileName
Answer whether a file or directory whose name is aFileName is included in the receiver.

isEmpty
Answer whether there are any files in the receiver.

statusOf: aFileName
Answer the status of aFileName without opening it.

xerox file compatability

findKey: aFileName
Answer an instance of the file class which represents a file with the name aFileName.

Class Methods

instance creation

currentDirectory
Answer an instance of me representing the current directory.

directoryFromName: fileName **setFileName:** aBlock
Answer the file directory implied from the designator fileName. This directory contains the designated file. Evaluate the block with only the file name portion of the designator.

directoryNamed: aString
Answer an instance of me whose name is aString. Answer Disk if aString matches Disk.

fileNamed: aFileName
Not appropriate for a FileDirectory.

Rationale

FileDirectory is a representation of a directory on the disk. It provides protocol for looking at the files in a directory.

Discussion

Disk is a global variable, an instance of **FileDirectory**. In the standard image **Disk** is set to ".", the current directory. This means that **Disk** "floats" to whatever directory you are in when you invoke Smalltalk. The Smalltalk home directory concept is used throughout file creation and referencing methods. There is example

code in the System Workspace for changing the value of **Disk**.

Class Protocol

Instance creation provides several ways to create an instance of **FileDirectory**. Instances can be created from the current directory, from the directory implied in a full path name, or by naming a directory. The inherited **fileNamed:** method is intercepted.

Instance Protocol

Accessing methods answer the directory name, answer a collection of the file names in the receiver, answer the **name** instance variable, and answer false to indicate that version numbers are not supported. Complete path means begin at root and fully specify the path. Full name means the name relative to **Disk** if **Disk** is part of the path, or the complete path relative to root (/) if **Disk** is not in the file's path.

Adding has one method which creates a new file with the specified name.

Enumerating has two methods which evaluate a block for all entries in the directory and a method, **filesMatching:**, that returns an array of file names matching an ambiguous file name — the ambiguous name provides the ability to use wildcard characters such as # or *.

File accessing methods perform many types of functions. You can check for invalid characters (less than ASCII 33 or greater than ASCII 126) in a file name and, optionally, remove the illegal characters. Two methods return true if a specified file name does not contain illegal characters; in addition, one of them checks that the file exists before answering true. One method answers the appropriate class of Smalltalk objects representing files in these directory objects — this allows some independence for **FileDirectory**.

Several *file accessing* methods return instances of this class and **FileStream**. You can choose among methods that check whether a file exists before creating a stream on it, that further specifies that the stream be write-only, or one that creates the stream without testing for the file's existence. This message category also has methods to create a new file or directory and rename an existing file.

If you don't want to always fully qualify the path of a file (or files) that you access frequently, you can use the following method.

- Create an instance of **FileDirectory** on the directory which contains the file(s). Keep the instance of **FileDirectory** in your image.

- Send a message to the **FileDirectory** to create a **FileStream** on the file when you want to access it (e.g., aFileDirectory oldFile: nameOfFile). The **FileStream** will be created relative to the path of the **FileDirectory.**

The above approach can be used for any directories you access often — just keep the instances of **FileDirectory** in your image.

File copying contains one method which appends the contents of a specified file to another file and one method which copies the contents of one file to another file.

Removing methods let you remove a disk file from the directory and, optionally, specify a block to be evaluated if the disk file does not exist.

Testing methods check whether a disk file exists, whether the directory is empty, and answer an instance of the file status class for your operating system.

The single *xerox file compatibility* method, **findKey:** calls **fileClass** and returns an instance of the class that method specifies. This method is retained for backward compatibility.

Examples

The following example can be executed in a workspace. If you do so, you might want to change the ambiguous file name to yield files you actually want to file in. For ease of use, a file list is better for filing in one file at a time. If you want a group of files to file in, the example code is a fast way to do it.

```
dir ← Disk directoryNamed: '/usr/lib/smalltalk/fileIn'.
(dir filesMatching: 'a*.st') do:
   [:eachFile | (dir file: eachFile) fileIn]
```

The global **FileDirectory, Disk,** is sent an *instance creation* message with the full path name as an argument. The instance is assigned to *dir*. An *enumerating* message, **filesMatching:**, is sent to *dir* with an ambiguous file name as the argument. For each file whose name begins with *a* and has the extension *.st*, an instance of **FileStream** is created. A **FileDirectory** *file accessing* message, **file:**, creates the instances. Each instance is sent the message **fileIn**, which incorporates the contents of the file into the image.

Related Classes

You are probably familiar with the results of some of this class' protocol, because **FileDirectory** is used in **FileLists.** Look in your System Workspace under "Create File System" for example code using **FileDirectory** and **FileStream.**

```
ExternalStream subclass: #FileStream

    instanceVariableNames:      'name directory mode fileDescriptor filePosition
                                fileMode '
    classVariableNames:         'OpenFileStreams '
    poolDictionaries:           ''
    category:                   'OS-Streams'
```

Summary

Instances of **FileStream** represent stream interfaces to mass storage files. Read and write data are buffered to minimize operating system calls. The buffer size is arbitrary; it need not reflect the physical disk block size. The buffer is either a **ByteArray** or a **String** depending on Smalltalk's view of the data. Characters and bytes are stored identically in mass storage.

Instance Variables

directory <FileDirectory>

This instance variable represents the directory containing the file, nil if unknown.

fileDescriptor <SmallInteger>

This instance variable represents the file identifier assigned by the operating system, or nil if the file is not open.

fileMode <Symbol>

This instance variable indicates the current permissions (#ReadOnly, #ReadWrite, or #WriteOnly) of this fileDescriptor. The value of fileMode reflects the last file activity in the file (e.g., #ReadWrite if the file was written to and then read). It can be #WriteOnly if FileStream opens the file write only or if the file is created by methods in this class.

filePosition <Integer>

This instance variable indicates the position of the operating system file pointer in the file.

mode <Symbol>

This instance variable indicates the intended reading/writing mode (#ReadOnly or #ReadWrite), nil if unknown.

name <String>
This instance variable identifies the file within a directory.

Inherited Instance Variables

collection <ByteArray> or <String>
This inherited instance variable is the buffer for either reading or writing data.

position <SmallInteger>
This inherited instance variable represents the current position in the buffer.

readLimit <SmallInteger>
This inherited instance variable represents the maximum position before the buffer is filled. When the buffer is empty, both position and readLimit are 0.

writeLimit <SmallInteger>
This inherited instance variable represents the maximum position before buffer is flushed.

Class Variables

OpenFileStreams <OrderedCollection>
All open FileStreams are listed here.

Instance Methods

accessing

contentsOfEntireFile
Read all of the contents of the receiver.

next
Answer the next character (or byte) from the receiver. Answer nil if at the end of the receiver's file.

next: anInteger
Answer the next anInteger bytes from the receiver.

next: anInteger **into:** aCollection
Copy the next anInteger bytes from the receiver into aCollection. If aCollection has word-sized elements, each element is filled with byte-sized numbers. Answer aCollection.

nextPut: aCharacterOrByte
Place the character or byte in the buffer and return that character. If the buffer is full, flush the buffer. If the file is not writable, make it writable. Call nextPut: again.

nextPutAll: aCollection
Write the elements of aCollection onto the receiver. If aCollection will fit in the receiver's buffer then buffer it, otherwise, write it directly to the receiver's file. If aCollection is not a String or ByteArray (a Set of Characters, for example) write each of its elements individually.

nextPutAll: aCollection **startingAt:** startIndex
Append the elements of aCollection, if it is of an appropriate type, onto the receiver starting at startIndex. Answer aCollection.

nextPutAll: aCollection **startingAt:** startIndex **to:** stopIndex
Append the elements of aCollection, if it is of an appropriate type, onto the receiver starting at startIndex and stopping at stopIndex. Answer aCollection.

size
Answer the size of the receiver's file in characters (or bytes).

converting

asFileDirectory
Return the file directory representing the receiver.

copying

copy
Answer a copy of the file with a nil file descriptor.

editing

edit

Create and schedule a FileModel on the contents of the receiver. The label of the view is the name of the receiver.

file accessing

description

Answer a String describing the receiver's file.

directory

Answer the directory that contains the receiver.

fileName

Answer the name of the receiver's file.

fullName

Answer the full path name of the file represented by the receiver.

name

Answer the name of the receiver's file.

remove

Remove the receiver from its parent directory. Discard the mass storage associated with the receiver.

rename: newFileName

Change the name of the receiver to newFileName.

file modes

binary

Set the receiver's file to be buffered in binary mode. Copy any already buffered data to the new buffer, a ByteArray.

readOnly

The receiver will be used for reading only.

readWrite

The receiver will be used for reading and writing. Do not backup on first write.

readWriteShorten
Same as readWrite. We don't support the shorten operation, so senders must explicitly close to truncate the file at the current position.

text
Set the receiver's file to be buffered in text mode. Copy any already buffered data to the new buffer, a String.

writeShorten
Same as readWrite. We don't support writeOnly, or the shorten operation. Senders must explicitly close to truncate the file at the current position.

file status

close
Disassociate the receiver with its file in mass storage. If write data are buffered, flush the buffer. If read data are buffered, discard the data and adjust filePosition to reflect the loss.

fill Fill the read buffer, collection, with data from mass storage. Flush the write buffer if required.

flush
Flush the output buffer, collection, to mass storage. Do nothing if the buffer is empty or is not an output buffer.

shorten
This operation is preserved for historical reasons. The structure of the existing file system does not encourage use of this file truncating method. The native use of backup files is suggested instead.

file testing

exists
Answer whether the file represented by the receiver exists in mass storage.

isBackup
Return true if the name of the file associated with this FileStream follows the convention for naming backup files.

isBinary
Answer whether the receiver is reading binary bytes (as opposed to characters).

isDirectory
Answer true if the receiver is a directory.

isOpen
Answer whether the receiver is open, that is, if the file represented by the receiver has been located and a file descriptor assigned.

isReadable
Answer whether it is possible to read from the receiver.

isStandard
Answer true if the receiver represents one of the standard descriptors — in, out, or error.

isText
Answer whether the receiver is reading characters (as opposed to binary bytes).

isWritable
Answer whether it is possible to write on the receiver.

fileIn-Out

fileIn
Guarantee fileStream is readOnly before fileIn for efficiency and to eliminate remote sharing conflicts.

fileOutChanges
Append to the receiver a description of all system changes.

printOutChanges
Print to the receiver a human-readable description of all system changes.

nonhomogeneous positioning

padTo: bsize
Skip to next boundary of bsize characters, and answer how many characters were skipped.

padTo: bsize **put:** aCharacterOrByte
Pad using the argument, aCharacterOrByte, to the next boundary of bsize characters, and answer how many characters were written.

positioning

position
Answer the position of the receiver in characters (or bytes) from its beginning. Compute this from the file's physical position considering any read ahead (reflected in readLimit) and the current position in the buffer.

position: anInteger
Position the receiver to start reading (or writing) at an offset of anInteger characters (or bytes) from the beginning of the file. Flush buffered output; discard (wastefully) buffered input. Position the receiver's file, if open.

reset
Set position to beginning of file.

setToEnd
Set position to end of file.

skip: byteCount
Advance position by byteCount. A negative byteCount will backspace. Preserve buffered read data if possible; always flush buffered write data.

printing

printOn: aStream
If the receiver is a file in the Disk directory, print with its path name relative to Disk. Otherwise, print its full path name.

testing

atEnd
Answer true if current position is >= end of file position. Fill the input buffer if necessary to access the next character.

xerox file compatability

asFileStream
Answer the file stream representing the receiver.

file Answer the file representing the receiver.

Class Methods

class initialization

initialize
Make a new collection for holding open files.

external references

closeExternalReferences
Close all open instances of the receiver and its subclasses. Use the bypassClose message to bypass the Smalltalk file tracking system. Only subtasks (child tasks) should execute this method.

releaseExternalReferences
Close all open instances of the receiver and its subclasses.

instance creation

fileNamed: aString
Answer a FileStream on an old or new file designated by aString. Do not (yet) try to open.

fileNamed: aString **in:** aDirectory
Answer a FileStream on an old or new file designated by aString and located in aDirectory.

newFileNamed: aString
Answer a FileStream on the new file designated by aString. Go ahead and create the file to be sure we can.

newFileNamed: aString **in:** aDirectory
Answer a FileStream on the new file designated by aString and located in aDirectory. Go ahead and create the file to be sure we can.

oldFileNamed: aString
Answer a FileStream on the old file designated by aString. Go ahead and open the file to be sure we can.

oldFileNamed: aString **in:** aDirectory
Answer a FileStream on the old file designated by aString and located in aDirectory. Go ahead and open the file to be sure we can.

oldWriteOnlyFileNamed: aString
Answer a FileStream on the old file designated by aString. Go ahead and open the file for writing to be sure we can.

oldWriteOnlyFileNamed: aString **in:** aDirectory
Answer a FileStream on the old file designated by aString and located in aDirectory. Go ahead and open the file for writing to be sure we can.

Rationale

FileStream provides a buffered streaming interface to mass storage files. The interface keeps the number of system calls to access files in the operating system to a minimum. It reopens files automatically (sources and changes files after a snapshot, for example). **FileStreams** are positionable, unlike pipe streams.

Discussion

Class Protocol

Class initialization contains the **initialize** method to assign values to the class variables. It would be used when you are working with a different operating system, for example, or want to change the size of the buffer.

External references methods enable you to close all open file streams or the streams with the file descriptors of stdin, stdout, and stderr. The method **closeExternalReferences** should only be sent by instances of **Subtask** to bypass the file tracking system and close all open file streams. It is important to remember that file descriptors are a limited resource, so file streams should be closed when they are no longer needed. Closing them releases their file descriptor for use in a new instance of **FileStream** or another object that requires an operating system file descriptor. The instance of **FileStream** will still exist after it is closed, and it can be reopened to the same state it was in when it was closed (i.e., at the same filePosition with the same mode and fileMode).

Instance creation methods return an instance of a stream interface to existing files and new files. You have the option of using a message beginning 'file' which creates the stream without opening its associated file; 'file' methods automatically create backup copies of existing files. 'New file' methods determine whether the specified file exists; if it does, a notifier displays to give you the option to "proceed" to rename the existing file as a backup. 'Old file' methods determine whether the specified file exists; if it does, a notifier displays to give you the option to "proceed" to create the file. Two methods will open an old file for writing only. They determine whether the specified file exists; if it does, a notifier displays to give you the option to "proceed" to create the file.

Instance Protocol

Some of the contents of the *accessing* message category is determined by what **FileStream** inherits from the **Stream** hierarchy. These methods are necessary to appropriately reimplement certain methods for this class. *Accessing* methods read and return the entire contents of a file, return the next one or more characters or bytes in a file, write one or more characters or bytes to the file, and return the effective size of the file in mass storage, were you to close the file at that point. The

method **size** includes data in the buffer which would be written to the file via **flush** if you sent the **close** message when the file was open for writing.

The *converting* method returns an instance of **FileDirectory** with the complete path name of the receiver.

The *editing* method creates a view containing the file contents for editing.

File accessing methods return information about the file obtained from instance variables or from a system call. **FullName** returns the full path of the receiver's directory concatenated with the receiver's name or a name relative to **Disk**. You can also remove or rename a file.

File modes methods allow you to set the **mode** instance variable to #ReadWrite or #ReadOnly. The method **binary** replaces the **collection String** with a **ByteArray**. The method **text** does the inverse of **binary**. Two 'shorten' methods call **readWrite** because the shorten operation is not supported. To shorten a file you must close it when the file is at the desired position.

File status methods close a file after flushing the buffer, fill the buffer for reading, flush the write data in the buffer to the file, and shorten the file. The **shorten** method is retained for backward compatibility — its use is not recommended.

File testing methods answer whether a file exists in mass storage, and returns true or false for the following file attributes: is a backup file, is binary, is text, is open, is readable, is writable, is a directory, and is a standard stream (in, out, or error known by file descriptors 0, 1, and 2).

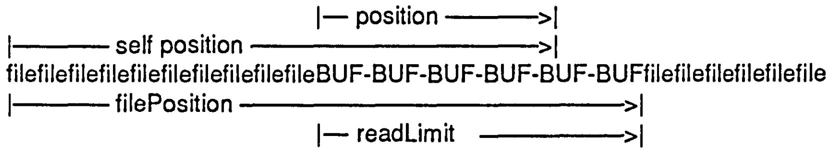
FileIn-Out methods incorporate the contents of an external file into the image, and enable you to file out or print out changes from the current project.

Nonhomogeneous positioning methods implement the inherited methods of the same name. The methods pad the file to a specified boundary with a specified character/byte or skip over characters or bytes in the file to the next specified boundary. This message category name is used for compatibility with protocol inherited from **ExternalStream**, although *nonhomogeneous* does not apply here.

Positioning methods answer the current location of the last character/byte conceptually read or written (buffered data are counted) or set the file pointer. The file pointer can also be set to the beginning or end of the file. The method **skip:** moves the **FileStream** pointer forward or backward and, if necessary, moves the operating system file pointer also. The **position:** method sets the file pointer to the specified location relative to the beginning of the file.

The following diagram describes the relationship of positions and the **FileStream** buffer. The instance variable **position** denotes the position within the buffer. The instance variable **filePosition** denotes the operating system file pointer associated with the file descriptor. Sending the message **position** returns the position the application thinks it should be at, taking into account unread data in the buffer. The instance variable **readLimit** describes the size of the buffer.

Relation of Position to the Buffer and the File in Mass Storage



Printing has one method that prints a representation of the instance on a stream. The output would resemble 'a FileStream on 'testFile''. The menu selection "printIt" calls **printOn:**.

Testing has one method that answers whether the end-of-file has been reached.

Xerox file compatibility methods return the receiver. They are present for compatibility with earlier implementations which distinguished between files and file streams.

Examples

The following code can be executed in a workspace.

```
f ← Disk file: 'testFile'.
f nextPut: $A.
f space.
f nextPutAll: 'file for reading '.
f cr.
f tab; nextPutAll: 'and writing'; cr.

f reset.
first ← f next.
all ← f contentsOfEntireFile.
Transcript cr; show: all.
```

```
Disk removeKey: 'testFile'.
```

First, the *file accessing* message **file:** is sent to the global **FileDirectory, Disk.** An instance of **FileStream** on 'testFile' is created and assigned to *f*. *Accessing* message **nextPut:** writes the character $A and **nextPutAll:** writes the string to 'testFile' via *f* (actually, the data are buffered and written to the file before it is closed). **WriteStream** *character writing* messages are used to put a space, a tab, and a carriage return in the file. The message **reset** sets the file pointer to the beginning of the file. *Accessing* messages **next** and **contentsOfEntireFile** return a character and the contents of the entire file, respectively. The entire file contents, *all*, are displayed in the System Transcript. Finally, 'testFile', is removed from the global **FileDirectory, Disk.** The output in the System Transcript looks like this:

```
A file for reading
   and writing
```

In the System Workspace under "Create File System" are several examples of **FileStream** and **FileDirectory** being used by the system. In a workspace you can execute `SourceFiles inspect` and see the array of two **FileStreams** on the sources file and changes file. You can further inspect the elements of the array to see the values of their instance variables.

Related Classes

Subclasses:

FileDirectory

Other classes of interest are

- the **PipeStream** hierarchy, and
- **ExternalStream**, the superclass, has useful methods for accessing different data types in *nonhomogeneous accessing*.

```
ExternalPointerData subclass: #FixedSizeExternalPointerData

    instanceVariableNames:      ''
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

FixedSizeExternalPointerData is an abstract class for a non-Smalltalk data structure containing machine pointers. Each subclass defines a particular data structure whose size is fixed. The concrete classes representing non-Smalltalk data structures are used to pass information between Smalltalk and the operating system, for example, when making system calls.

Instance Methods

accessing

numberOfPointers
Return the number of pointers imbedded in the structure represented by the receiver.

pointersSize
Answer the size of the pointers array of the receiver.

```
FixedSizeExternalPointerData class

    instanceVariableNames:      'sizeInBytes prototype '
```

FixedSizeExternalPointerData class — Instance Variables

sizeInBytes <Integer>
The size of the structure in bytes.

prototype <aSubclass>
An instance of the concrete subclass which has the pointers array initialized.

Class Methods

accessing

numberOfPointers
Return the number of pointers imbedded in the structure represented by the receiver.

pointersSize
Answer the size of the pointers array.

sizeof
Answer the number of bytes in the data section of any instance.

instance creation

new
Return a new instance of the receiver.

Rationale

Since all external data classes either do or do not contain pointers, the **ExternalData** hierarchy splits into **ExternalBinaryData** and **ExternalPointerData**. **FixedSizeExternalPointerData**, a subclass of **ExternalPointerData**, is an abstract class which implements protocol common to all of its concrete subclasses which, by definition, contain pointers and are a fixed size. Objects which are *not* a fixed size but contain pointers, such as an array of pointers or an array of structures which *may* contain pointers, are direct subclasses of **ExternalPointerData**.

Inherited protocol designated as subclass responsibility by **ExternalData** is implemented. All subclasses of this branch of the external data hierarchy have two class instance variables. **SizeInBytes** is defined here as the number of bytes in the data area of all instances of a subclass. **Prototype** is defined here as an instance of a subclass with its **pointers** array initialized.

Discussion

You will find it helpful to read the "Discussion" under **ExternalPointerData** for an explanation of the instance and class variables of concrete subclasses of this class.

Class Protocol

This protocol is present to be inherited by concrete subclasses. Since this is an abstract class, there is no instance creation protocol and no messages should be sent to this class.

Accessing methods answer the number of pointers in the structure, the number of bytes of data in the structure, and the number of elements in **pointers**.

Instance creation defines **new** to return a copy of the **prototype**.

Instance Protocol

Accessing has one method which calls the class method **pointersSize** to answer the number of elements in **pointers** and another method which calls the class method to return the number of pointers.

Adding Classes to this Hierarchy

If you are adding a subclass of **FixedSizeExternalPointerData** and the conventions described in this manual under **ExternalPointerData** are unclear to you, the best idea is to find a structure class similar to the structure you are adding and model your class on the similar one. Your new class' protocol must include a class initialization method which initializes the class variables, **sizeInBytes**, and **prototype**. Remember to execute the **initialize** method for the new class.

Related Classes

Subclasses:

Iovec
Msghdr

You might also want to look at **ExternalPointerData** and **ExternalData**.

```
ExternalBinaryData variableByteSubclass: #Inaddr

    instanceVariableNames:      ''
    classVariableNames:         'AddrDataIndex B1DataIndex B2DataIndex
                                B3DataIndex B4DataIndex W1DataIndex
                                W2DataIndex '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

Inaddr provides accessing protocol for the following C structure.

```
struct in_addr {
   union {
                  struct { u_char s_b1, s_b2, s_b3, s_b4; } S_un_b;
                  struct { u_short s_w1, s_w2; } S_un_w;
                  u_long S_addr;
   } S_un;
#define    s_addr    S_un.S_addr          /* can be used for most
                                           tcp and ip code */
#define    s_host    S_un.S_un_b.s_b2     /* host on imp */
#define    s_net     S_un.S_un_b.s_b1     /* network */
#define    s_imp     S_un.S_un_w.s_w2     /* imp */
#define    s_impno   S_un.S_un_b.s_b4     /* imp # */
#define    s_lh      S_un.S_un_b.s_b3     /* logical host */
}
```

The structure is referred to under *inet(4N)* in the manual *UTek Command Reference, Volume 2*. The structure is found in *netinet/in.h*. Protocol is provided to set the values from an integer and from a string.

Class Variables

AddrDataIndex

B1DataIndex

B2DataIndex

B3DataIndex

B4DataIndex

W1DataIndex

W2DataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **B3DataIndex** holds the offset of the "s_b3" field.

Instance Methods

accessing

addr
Return the value of the internet address as an integer.

addr: anInteger
Assign the argument, anInteger, as the internet address.

converting

asString
Return the value of the internet address as a string.

printing

printOn: aStream
Print the receiver in internet format on aStream.

private

fromString: anInternetAddress
Assign anInternetAddress string (in dot notation) to the receiver.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

addr: anInteger
Return an instance constructed from the internet address, anInteger.

fromString: anInternetAddressString
Return an instance constructed from the internet address, anInternetAddressString.

Rationale

This class is the C structure which holds an internet address. It holds a network address, and is used by the class **SockaddrIn**. The structure is used in support of the following UTek system calls:

accept(2)
bind(2)
connect(2)
getpeername(2)
getsockname(2)
recvfrom(2)
sendto(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
ExternalBinaryData variableByteSubclass: #IntegerPointer

    instanceVariableNames:      ''
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

IntegerPointer provides creation and accessing protocol for integer pointers. Integer pointers are defined in C with the type-specifier 'int *'. A four-byte buffer stores integer values.

Instance Methods

accessing

integer
Return the signed integer that the receiver represents.

integer: anInt
Assign the value, anInt, to the receiver.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Define the size of the structure.

instance creation

integer: anInt
Return an instance whose value is the machine integer for anInt.

Rationale

An argument to a system call that needs an integer pointer uses an instance of class **IntegerPointer**. An example use is in *getpeername(2)*, where the size of the remote machine name returned is an instance of this class.

```
ExternalPointerData subclass: #Iovec

    instanceVariableNames:     ''
    classVariableNames:        'BaseDataIndex BasePointerIndex LenDataIndex
                               '
    poolDictionaries:          ''
    category:                  'OS-Parameters'
```

Summary

Iovec provides creation and accessing protocol for the following C structure.

```
struct iovec {
    caddr_t    iov_base;
    int        iov_len;
}
```

Each iovec entry specifies the base address and length of an area in memory. The structure is documented under *read, readv(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

BaseDataIndex

BasePointerIndex

LenDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **BaseDataIndex** holds the offset of the "iov_base" field.

For fields with a pointer data type, a class variable is created for the index of that pointer in the structure; the name is constructed from the field name, stripped of its prefix, with the string 'PointerIndex' appended. For example, the class variable **BasePointerIndex** holds the index "1" because it is the first pointer in the iovec structure.

Instance Methods

accessing

base
Return the value of the structure field named base.

base: aString
Assign the argument aString to the structure field named base, and assign aString's size to the structure field named len.

len Return the value of the structure field named len.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables, define the size of the structure, and initialize the prototype.

instance creation

base: aString
Return an instance with the values of the fields assigned.

Rationale

The iovec C structure is used in support of the following UTek system calls:

readv(2)
writev(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
ExternalBinaryData variableByteSubclass: #Itimerval

    instanceVariableNames:   ''
    classVariableNames:      'IntervalDataIndex IntervalSecDataIndex
                             IntervalUsecDataIndex ValueDataIndex
                             ValueSecDataIndex ValueUsecDataIndex '
    poolDictionaries:        ''
    category:                'OS-Parameters'
```

Summary

Itimerval provides creation and accessing protocol for the following C structure.

```
struct itimerval {
    struct timeval   it_interval;   /* timer interval */
    struct timeval   it_value;      /* current value */
}
```

The structure is documented under *getitimer(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

IntervalDataIndex

IntervalSecDataIndex

IntervalUsecDataIndex

ValueDataIndex

ValueSecDataIndex

ValueUsecDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **IntervalDataIndex** holds the offset of the "it_interval" field.

When a field in a structure is another C structure, separate class variables are created for each field in the other structure. For example, the class variable **IntervalSecDataIndex** holds the offset of the "sec" field of the "it_interval" structure.

Instance Methods

accessing

interval
Return the value of the structure field named interval.

interval: aTimeval
Assign the argument, aTimeval, to the structure field named interval.

interval: aTimeval **value:** anotherTimeval
Assign values to all the fields of the structure.

intervalSec
Return the sec field of the structure at the field named interval.

intervalUsec
Return the usec field of the structure at the field named interval.

value
Return the value of the structure field named value.

value: aTimeval
Assign the argument, aTimeval, to the structure field named value.

valueSec
Return the sec field of the structure at the field named value.

valueUsec
Return the usec field of the structure at the field named value.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

interval: aTimeval **value:** anotherTimeval
Return an instance with the values of the fields assigned.

Rationale

The structure is used in support of the following UTek system calls:

getitimer(2)
setitimer(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
ExternalBinaryData variableByteSubclass: #Ltchars

    instanceVariableNames:    ''
    classVariableNames:       'DsuspcDataIndex FlushcDataIndex
                              LnextcDataIndex RprntcDataIndex
                              SuspcDataIndex WerascDataIndex '
    poolDictionaries:         ''
    category:                 'OS-Parameters'
```

Summary

Ltchars provides accessing protocol for the following C structure.

```
struct ltchars {
    char   t_suspc;    /* stop process signal */
    char   t_dsuspc;   /* delayed stop process signal */
    char   t_rprntc;   /* reprint line */
    char   t_flushc;   /* flush output (toggles) */
    char   t_werasc;   /* word erase */
    char   t_lnextc;   /* literal next character */
}
```

The structure is documented under *tty(4)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

DsuspcDataIndex

FlushcDataIndex

LnextcDataIndex

RprntcDataIndex

SuspcDataIndex

WerascDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **DsuspcDataIndex** holds the offset of the "t_dsuspc" field.

Instance Methods

accessing

dsuspc
Return the value of the structure field named dsuspc.

dsuspc: aCharacter
Assign the argument, aCharacter, to the structure field named dsuspc.

flushc
Return the value of the structure field named flushc.

flushc: aCharacter
Assign the argument, aCharacter, to the structure field named flushc.

lnextc
Return the value of the structure field named lnextc.

lnextc: aCharacter
Assign the argument, aCharacter, to the structure field named lnextc.

rprntc
Return the value of the structure field named rprntc.

rprntc: aCharacter
Assign the argument, aCharacter, to the structure field named rprntc.

suspc
Return the value of the structure field named suspc.

suspc: aCharacter
Assign the argument, aCharacter, to the structure field named suspc.

suspc: sCharacter **dsuspc:** dCharacter **rprntc:** rCharacter **flushc:** fCharacter **werasc:** wCharacter **lnextc:** lCharacter
Assign values to all the fields of the structure.

werasc
Return the value of the structure field named werasc.

werasc: aCharacter
Assign the argument, aCharacter, to the structure field named werasc.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

default
Return an instance containing the default characters.

Rationale

The structure is used in support of the following UTek system call:

ioctl(2)

Related Classes

UTekSystemCall implements the system call listed above.

```
ExternalPointerData subclass: #Msghdr
    instanceVariableNames:    ''
    classVariableNames:       'AccrightsDataIndex AccrightslenDataIndex
                              AccrightsPointerIndex IovDataIndex
                              IovlenDataIndex IovPointerIndex NameDataIndex
                              NamelenDataIndex NamePointerIndex '
    poolDictionaries:         ''
    category:                 'OS-Parameters'
```

Summary

Msghdr provides creation and accessing protocol for the following C structure.

```
struct msghdr {
    caddr_t        msg_name;          /* optional address */
    int            msg_namelen;       /* size of address */
    struct iovec   *msg_iov;          /* scatter/gather array */
    int            msg_iovlen;        /* # elements in msg_iov */
    caddr_t        msg_accrights;     /* access rights sent/received */
    int            msg_accrightslen;
}
```

The structure is documented under *recvmsg(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

AccrightsDataIndex

AccrightslenDataIndex

AccrightsPointerIndex

IovDataIndex

IovlenDataIndex

IovPointerIndex

NameDataIndex

NamelenDataIndex

NamePointerIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from the field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **AccrightsDataIndex** holds the offset of the "msg_accrights" field.

For fields with a pointer data type, a class variable is created for the index of that pointer in the structure; the name is constructed from the field name, stripped of its prefix, with the string 'PointerIndex' appended. For example, the class variable **AccrightsPointerIndex** holds the index "3" because it is the third pointer in the msghdr structure.

Instance Methods

accessing

accrights
Return the value of the structure field named accrights.

accrights: aString
Assign the argument, aString, to the structure field named accrights, and assign aString's size to the structure field named accrightslen.

accrightslen
Return the value of the structure field named accrightslen.

iov Return the value of the structure field named iov.

iov: anIovecArray
Assign the argument, anIovecArray, to the structure field named iov, and assign anIovecArray's size to the structure field named iovlen.

iovlen
Return the value of the structure field named iovlen.

name
Return the value of the structure field named name.

name: aString
Assign the argument, aString, to the structure field named name, and assign aString's size to the structure field named namelen.

name: aString **iov:** anIovecArray **accrights:** anotherString
Assign values to all the fields of the structure.

namelen
Return the value of the structure field named namelen.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables, define the size of the structure, and initialize the prototype.

instance creation

name: aString **iov:** anIovecArray **accrights:** accString
Return an instance with the values of the fields assigned.

Rationale

The msghdr C structure is used in support of the following UTek system calls:

recvmsg(2)
sendmsg(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
PipeReadStream variableSubclass: #OSFilter

    instanceVariableNames:      'task stdin stderr '
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Streams'
```

Summary

An **OSFilter** provides bi-directional communication with a utility program, which may be either an operating system utility or a user supplied program. It understands accessing protocol for both **PipeReadStream** and **PipeWriteStream.** Writing to an **OSFilter** causes its task to receive standard input. Reading an **OSFilter** causes the task to supply its standard output. The utility is not required to read standard input (in which case writing protocol has no effect), nor to write standard output or standard error (in which case reading protocol never shows data available). Note that the utility may not produce output immediately, depending on how it buffers its output.

Instance Variables

stderr <PipeReadStream>

The stream that collects the utility's standard error output.

stdin <PipeWriteStream>

The stream that supplies the utility with standard input.

task <Subtask>

The system utility being used as a filter.

Instance Methods

initialize-release

initializeErrorOn: aPipe

Initialize the error output side of the filter stream with the supplied pipe which contains an open read file descriptor.

initializeErrorOnFd: fd

Initialize the error output of the filter stream with the supplied file descriptor, which is assumed to be the reading end of a Pipe.

initializeInputOn: aPipe
Initialize the input side of the filter stream with the supplied pipe which contains an open write file descriptor.

initializeInputOnFd: fd
Initialize the input side of the filter stream with the supplied file descriptor, which is assumed to be the writing end of a Pipe.

accessing

inputBinary
Set the mode of this OSFilter to binary.

inputText
Set the mode of this OSFilter to text.

nextError
Answer a String containing the next line from the error stream.

nextPut: aCharacter
Place aCharacter in the task's input stream.

nextPutAll: aCollection
Place all the data in aCollection in the task's input stream.

outputBinary
Set the OSFilter's output to be buffered in binary mode.

outputText
Set the OSFilter's output to be buffered in text mode.

opening-closing

abort
Clean up. Close all PipeStreams and terminate the task.

close
Close the standard input to the task. Return whatever might be produced in standard output. Close standard output and standard error and terminate the task.

closeError
Close the error output from the task. Return whatever might have been there.

closeInput
Close the standard input to the task. Normally this causes the task to terminate.

openOn: aCommand **withArguments:** aCollection
Initialize the filter by opening input and output streams and starting the task.

positioning

flush
Chase something through the filter with multiple line terminators.

flushWith: aCharacter **blockSize:** count
Chase something through the filter with count occurrences of the given character.

printing

printOn: aStream
Place in aStream a String that describes the program and arguments of an OSFilter. If the task is inactive, so state.

testing

isActive
Is this filter's task active?

Class Methods

examples

example
Use sed and cpp to extract key-value pairs from the C header file, 'errno.h'.

valuesFromHeader: aHeaderFile
Use sed and cpp to extract a Dictionary of key-value pairs from a C header file. This may take a few minutes.

instance creation

openOn: aCommand
Create a new OSFilter, with its input and output streams directed to/from aCommand.

openOn: aCommand **withArguments:** aCollection
Create a new OSFilter, with its input and output streams directed to/from aCommand. Pass aCollection of arguments to aCommand upon execution.

Rationale

OSFilter was designed as a general purpose tool for programmers to be able to use operating system utilities or user supplied programs from within Smalltalk.

Background

Filters

Filters are programs which process a character stream — usually, they modify the stream. There are some filters, such as *cat*, which do not modify their input, but they are not used as examples here. The filter has an input side and an output side — in UTek the reading is usually done from standard input.

In building the operating system interface to Tektronix Smalltalk, it quickly became apparent that many of the utility programs available could find use within Smalltalk. Without **OSFilter**, Smalltalk could receive output from an operating system program using the **UTekSystemCall** method **executeUtility:withArguments:**, but it couldn't provide input to the operating system program. The background menu item "OS Shell" would allow user interaction with various programs, but there was no simple means for a Smalltalk method to interact with a program.

Two-Way Communication

What was needed was a facility for bi-directional communication with a utility program. **OSFilter** accomplishes this by running a utility program with individual PipeStreams for the program's standard input and standard output. **OSFilter** is designed for portability across operating systems.

Examples

The following example illustrates the use of an instance of **OSFilter** to access the UTek program *wc*. This code can be executed in a workspace.

```
wc ← OSFilter openOn: '/bin/wc'.    "path for UTek only"
wc nextPutAll: 'How many lines, words and characters are there
in this string?'.
wc closeInput.
lines ← wc nextCNumber.
words ← wc nextCNumber.
characters ← wc nextCNumber.
wc close.
Transcript cr; show: lines printString, ' lines, ',
    words printString, ' words, and ',
    characters printString, ' characters are in the string.'
```

When the preceding code is executed in a workspace, this output appears in the System Transcript:

2 lines, 11 words, and 62 characters are in the string.

An Example from the Image

A more complex example of an **OSFilter** application is found in the **OSFilter class** method **valuesFromHeader:** under the *examples* protocol.

```
valuesFromHeader: aHeaderFile

    "Use sed and cpp to extract a Dictionary of key-value pairs from a C
       header file. This may take a few minutes."
    "(OSFilter valuesFromHeader: '/usr/include/sys/max.h') inspect."

    | sed cpp key symbolTable |
    symbolTable ← Dictionary new: 128.
    "Open a sed filter that will extract just C preprocessor symbols from a
       given file."
    sed ← OSFilter openOn: '/bin/sed' withArguments: (OrderedCollection
       with: '-n'
       "A space and a tab are inside the square brackets below that appear empty."
       with:
       's/#[    ]*define[    ][    ]*\([A-Za-z_][0-9A-Za-z_]*\)[    ].*/\1/p'
       with: aHeaderFile).
    sed closeInput.
    "Open a C preprocessor filter, feed it the given file, and discard whatever
       that might produce."
    cpp ← OSFilter openOn: '/lib/cpp'.
    cpp nextPutAll: '#include "', aHeaderFile, '"'.
    cpp flush; nextAvailable.          "Discard uninteresting output."
    "Feed each symbol from sed into cpp, collect its output in a Dictionary."
    [sed atEnd]
       whileFalse:
          [key ← sed nextLine.
          key ~= '' ifTrue:
             [cpp nextPutAll: key.
             cpp flush.
             symbolTable at: (key copyUpTo: Character cr) asSymbol
                      put: cpp nextCNumber.
             cpp nextAvailable]].          "Discard the flush characters."
    sed close.
```

```
cpp close.
↑symbolTable
```

Here is a description of what the code is doing. The method **valuesFromHeader:** takes a ".h" file (a C header file) and finds all the non-argument defines in the form

```
#define    identifier    token-string
```

and creates dictionary entries with the defined identifier and its associated token-string, a CNumber. This method only creates dictionary entries for defined identifiers whose associated token-string is a valid C number, using the **PositionableStream** method **nextCNumber.**

The *sed* filter only extracts simple identifiers, not parameterized ones (with arguments). For example,

```
#define    sqr(x)    (x * x),
```

will not cause *sqr* to be entered in the dictionary.

First the dictionary called *symbolTable* is created. Then a *sed* **OSFilter** is run on the header file with the operating system *sed* command that causes only the defined identifiers to appear in the output stream, one per line. Input to *sed* is closed, causing all of the data from *sed* to become available. A *cpp* **OSFilter** is opened and given the header file as input. *Cpp* is flushed and a loop begins in which *sed* output is sent to *cpp*, one line at a time, and the identifier from *sed* is placed into the dictionary *symbolTable* with its associated CNumber from *cpp*. After each line from *sed* goes into *key* and then to *cpp*, *cpp* is flushed. *Cpp* buffers its output and we want it to deal with one identifier at a time. (**Flush** outputs multiple line terminators, so they must be discarded by sending *cpp* the message **nextAvailable.**) When the data from *sed* are exhausted the loop terminates, both filters are closed, and *symbolTable* is returned.

Related Classes

An instance of **Subtask** is the program, residing in the operating system, that is serving as the filter in the **OSFilter.** In addition to reading about **Subtask,** it will also be useful to look at the **Stream** hierarchy, in particular, **OSFilter's** superclass, **PipeReadStream.**

Pipe

```
Object subclass: #Pipe

    instanceVariableNames:     'readFd writeFd fileDescriptor '
    classVariableNames:        ''
    poolDictionaries:          ''
    category:                  'OS-Interface'
```

Summary

An instance of **Pipe** represents an operating system pipe in Smalltalk. Operating system pipes are unidirectional communication links, usually between Smalltalk and a child task in our context. A pipe has two file descriptors, one for each end of the pipe. One end of the pipe is for reading, the other for writing. A system buffer holds the contents of the pipe.

Communication between tasks is accomplished via a **PipeStream** which is opened using one of the two descriptors, depending on the direction. Two way communication between two tasks requires two pipes.

Instance Variables

fileDescriptor <Integer>
The read or write file descriptor used to access the parent's end of the pipe.

readFd <Integer>
The file descriptor used to access the reading end of the pipe.

writeFd <Integer>
The file descriptor used to access the writing end of the pipe.

Instance Methods

initialize-release

readFd: readingEnd **writeFd:** writingEnd
Initialize a new Pipe with the given reading and writing file descriptors.

release
Close both ends of the pipe.

accessing

fileDescriptor
Return the file descriptor to be used for streaming over data, either the readFd or the writeFd.

readFd
Return the file descriptor representing the read end of the pipe.

writeFd
Return the file descriptor representing the write end of the pipe.

copying

copy
Inappropriate for a Pipe.

opening-closing

closeRead
Close the read side of the pipe.

closeWrite
Close the write side of the pipe.

fileDescriptor: fd
Fd, which is either this Pipe's readFd or its writeFd, becomes the descriptor to be accessed for streaming over data.

mapReadTo: fd
Map the read side of the pipe to the specified descriptor fd.

mapWriteTo: fd
Map the write side of the receiver's pipe to the specified descriptor fd.

Class Methods

instance creation

new
Return an instance of Pipe.

readFd: readingEnd **writeFd:** writingEnd
Create a new Pipe, with the given reading and writing file descriptors.

Rationale

An instance of **Pipe** is a temporary holding place for file descriptors. Once a **Pipe**'s file descriptor is passed to a **PipeStream**, the **Pipe** is no longer needed, and should not be referenced.

Discussion

Instances of **Pipe** are created using either of two *instance creation* messages. The message **new** causes the operating system to return the pipe's file descriptors. The message **readFd:writeFd:** creates a new **Pipe** using file descriptors obtained by some other means (e.g., OS duplicateFd:).

To insure portability, instead of sending the message **new** to the class **Pipe,** instances should be created this way:

```
OS newPipe.
```

The *initialize-release* message **release** closes the reading and writing ends of the pipe. Closing pipes is often accomplished using the separate *opening-closing* messages **closeRead** and **closeWrite.**

Accessing messages return the values of a **Pipe**'s three instance variables.

Protocol exists for mapping ends of the pipe to arbitrary file descriptors. The methods **mapReadTo:** and **mapWriteTo:** make operating system calls to duplicate the argument, a file descriptor. Mapping allows communication between parent and child tasks.

Examples

The **OSFilter** method **openOn:withArguments:** contains several examples of typical pipe usage:

```
openOn: aCommand withArguments: aCollection

"Initialize the filter by opening input and output streams and
starting the task."

    | in out err |
    in ← OS newPipe.
    out ← OS newPipe.
    err ← OS newPipe.
    self initializeOn: out.
    self initializeInputOn: in.
    self initializeErrorOn: err.
```

```
self task: (Subtask
        fork: aCommand
        withArguments: aCollection
        standardIn: in
        standardOut: out
        standardError: err).
self task start isNil
    ifTrue:
        [self stdin close.
        self close.
        Transcript show: self closeError.
        self error: 'Cannot execute ', aCommand].
in closeRead.
out closeWrite.
err closeWrite
```

The three temporary variables *in, out,* and *err* each receive new instances of **Pipe** via the message **newPipe**. Newpipe is always sent to the current operating system call class (referred to by the global variable **OS**). The following initialization messages cause the **OSFilter** to get the proper file descriptors of the three **Pipes** — the **OSFilter** doesn't get the **Pipes** themselves. In the **Subtask** creation message, the new **Subtask** gets the file descriptors for the opposite ends of the three **Pipes**. Finally, the ends of the **Pipes** not needed by the **OSFilter** (i.e., those given to the **Subtask**) are closed. When this method completes, the three **Pipes** are no longer referenced and are garbage collected.

The method above is a succinct example of a portable system call, subtasking, and the use of pipes in the class **OSFilter**.

Related Classes

These classes are related to **Pipe**:

OSFilter
PipeStream
PipeReadStream
PipeWriteStream
Subtask

```
PipeStream subclass: #PipeReadStream

    instanceVariableNames:      'foundEnd '
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Streams'
```

Summary

PipeReadStreams are used for reading from an operating system pipe. Data are buffered from the pipe to avoid excessive reads. Protocol for reading all the data available from the pipe is provided by using the size of the pipe as an indicator of how much to read — reading more data than are available can result in blocking.

Instance Variables

foundEnd <Boolean>
True if the end of the pipe has been encountered.

Instance Methods

initialize-release

initializeOn: aPipe
Initialize this PipeStream with aPipe, which must contain an open read file descriptor.

initializeOnFd: fd
Initialize this PipeStream with the supplied open file descriptor, which must refer to a Pipe.

accessing

binary
Set the receiver's file to be buffered in binary mode. Copy any already buffered data to the new buffer, a ByteArray.

contentsOfEntireFile
Read all of the contents of the receiver. If the PipeReadStream buffer contains something, prepend it to the results.

next
Return the next character or byte from the PipeReadStream. Answer nil if at the end of the stream.

next: anInteger
Return the next anInteger bytes from the PipeReadStream. Return a ByteArray if the stream is binary, return a String if the stream is text.

next: anInteger **into:** aCollection
Copy the next anInteger bytes from the receiver into aCollection. Answer aCollection.

nextAvailable
Answer a collection containing all available data from the pipe. Answer an empty collection if no data are available. Answer nil if something is wrong with the pipe.

nextPut: anObject
Not apropriate for a PipeReadStream.

text
Set the receiver's file to be buffered in text mode. Copy any already buffered data to the new buffer, a String.

enumerating

doAvailable: aBlock
Evaluate aBlock for each object in the pipe.

doByLine: aBlock
Evaluate aBlock for each string ending with a line terminator in the pipe. Do not include the line terminator in the string.

positioning

skip: count
Discard the next count items in the stream.

testing

atEnd
Return true if the end was previously found or answer a guess based on the size of the contents of the pipe.

dataAvailable
Return true if there are any data ready to be read from the pipe.

isEmpty
Return true if the pipe contains no data.

Rationale

This class allows the reading of data from an operating system pipe. Using a **PipeReadStream**, a Smalltalk process can obtain data from a related process.

Discussion

The abstraction of a pipe stream is used outside of Smalltalk, for example, several C functions use a stream for reading, including fscanf, gets, and getc. Most users are familiar with the standard input stream, stdin.

Streams make accessing data easier, because they "know" the position where reading or writing can occur. It isn't necessary for you to keep track of how many characters have been read so that you can specify which character to read next. A stream knows when it has reached the end of its data and when it is empty.

Protocol

To create an instance of **PipeReadStream**, you should use an *instance creation* message inherited from the superclass, **PipeStream** — for example, send the message **openOn:** to **PipeReadStream.**

Initialize-release methods initialize appropriate instance variables and add this **PipeReadStream** to the class variable **OpenPipeStreams.**

Accessing methods set the **mode** instance variable and, if the mode was incorrect, copy the data into the correct type of buffer (ByteArray or String). Data in the stream can be read — the entire file, only the contents of the buffer, a single byte, or a specified number of bytes. Data from the stream can be read and returned in a **Collection.** The inherited message **nextPut:** is intercepted because this is a stream for reading.

Enumerating methods evaluate a block for each byte/character or each line in the stream.

Positioning allows you to skip over (discard) a specified number of bytes/characters in the stream.

Testing methods answer whether the end of the stream's data has been reached, whether there are data available for reading, and whether the pipe is empty.

Examples

The following method is found under **SystemDictionary** *initialize-release.*

```
startUp
    "Do all that is necessary to begin execution of a Smalltalk image."

    | stdin |
    ScheduledControllers restore.
    OS notNil ifTrue:
        [(OS status: 0) fileSize = 0
            ifFalse:
                [stdin ← PipeReadStream openOnFd: (OS duplicateFd: 0 with: 5).
                Transcript refresh; cr; cr; show: 'Filing in from: standard input'; cr.
                stdin fileIn; close].
        Subtask markAndSignalAll; install].
    Sensor cursorPoint: Display viewport topLeft
```

In addition to starting Smalltalk, the method above provides the ability for you to pass a file or "doIt" to Smalltalk from the command line when you invoke an image. It makes a system call to determine if the pipe for standard input (file descriptor = 0) has size 0. If there is anything in that pipe, an instance of **PipeReadStream** is created on standard input, but using a different file descriptor — the **duplicateFd:** message links the new stream with standard input. Next the "filing in" message is displayed in the System Transcript. Smalltalk then attempts to "file in" whatever is present in standard input. If the standard input is not in "file in" format, a syntax error will result. Typical fileIn files have names ending in *.st* or *.ws*. Here are two examples of command lines which pass arguments to Smalltalk:

```
echo "Transcript show: 'Your message here.'.!" | <smalltalk or imageName>

cd <smalltalk fileIn directory>
smalltalk < Clock.st
```

In the first example you would type `smalltalk` or the name of your image in the position indicated by the angle brackets (< >). The string you type as the argument to **show:** will be displayed in the System Transcript when the image comes up. In the second example, you first *cd* to the directory which contains the fileIn files. Without moving to that directory, you could accomplish the same thing by giving the full path of *Clock.st* on line two. The clock code will be filed in when the image comes up. The first example uses a shell pipe operator (|); the second example uses a shell I/O redirection operator ("<" for input).

Related Classes

Subclasses:

OSFilter

You will find it helpful to look at the following classes for a further understanding of this class:

- the **PipeStream** hierarchy,
- **Pipe**, and
- methods **nextLine** and **nextCNumber** in **PositionableStream.**

ExternalStream subclass: #PipeStream

instanceVariableNames:	'fileDescriptor mode '
classVariableNames:	'OpenPipeStreams '
poolDictionaries:	''
category:	'OS-Streams'

Summary

PipeStreams are stream-based interfaces to operating system pipes. **PipeStream** is an abstract class implementing protocol common to **PipeReadStream** and **PipeWriteStream**, such as opening, closing, and accessing methods.

Instance Variables

fileDescriptor <Integer>
The file descriptor by which the underlying pipe is accessed.

mode <Symbol>
Either #binary or #text, depending on what type of data is expected to be used on the underlying pipe.

Class Variables

OpenPipeStreams <OrderedCollection>
All open PipeStreams are listed here.

Instance Methods

initialize-release

initializeOnFd: anFd
Initialize the pipe stream with the supplied open file descriptor, which must refer to a Pipe.

accessing

binary
Set the mode of this PipeStream to binary.

bufferClass
Return the class that is used as a buffer for this PipeStream, depending on the data type which has been selected.

contents
Not appropriate for a PipeStream.

fileDescriptor
Answer the descriptor to be accessed for streaming over data.

fileDescriptor: fd
Store the argument as the descriptor to be accessed for streaming over data.

size
Return the size of the PipeStream contents as reported by the operating system.

text
Set the mode of this PipeStream to text.

copying

copy
Not appropriate for a PipeStream.

opening-closing

close
Disassociate this PipeStream with its underlying pipe. Make sure that the pipe exists, then make sure it is closed.

positioning

padTo: anInteger
Not appropriate for a PipeStream.

padTo: bsize **put:** aCharacter
Not appropriate for a PipeStream.

position: anInteger
Not appropriate for a PipeStream.

reset
Not appropriate for a PipeStream.

resetContents
Not appropriate for a PipeStream.

setToEnd
Not appropriate for a PipeStream.

skip: anInteger
Not appropriate for a PipeStream.

wordPosition: wp
Not appropriate for a PipeStream.

testing

atEnd
Is this PipeStream at the end of its data?

isBinary
Does this PipeStream contain binary data?

isText
Does this PipeStream contain textual data?

isValid
Does this PipeStream reference an open pipe?

Class Methods

class initialization

initialize
Initialize the class variable used for tracking open PipeStreams.

external references

numberOfExternalReferences
Return the number of references to open PipeStreams.

releaseExternalReferences
Close all the open PipeStreams known to Smalltalk.

instance creation

openOn: aPipe
Create a new PipeStream, streaming over aPipe.

openOnFd: fd
Create a new PipeStream, streaming over a Pipe, which is known by the open file descriptor fd.

Rationale

PipeStream is an abstract class whose subclasses are the primary means of communication with child tasks. Operating system pipes are usually implemented in

Smalltalk using **PipeStream.** An important function of this class is to intercept inherited methods which are inappropriate for a **PipeStream.** Pipes are not positionable, so **PipeStream** removes much of the *positioning* protocol from the inheritance chain via "self shouldNotImplement".

Background

Terminology

Here is a clarification of some of the terms used in the description of the classes in the **Stream** hierarchy, and in particular, the **PipeStream** and **FileStream** hierarchies.

File has a variety of meanings, depending upon your operating system. In the context of these classes, a file is a collection of data in external storage, referred to by a name. In this respect, files differ from operating system pipes, which are not named by the user. In Unix-like operating systems, the distinction between files and directories (and other non-file objects, as well) is blurred. The implementations of **FileStream** and **FileDirectory** maintain that ambiguity, to some extent. See those classes for additional information.

Pipes share a similarity with files in that both are holding places for data. The primary differences between the two are that pipes are transient (they do not reside in mass storage), they cannot be reopened once they are closed, and they are not positionable. Like files, the operating system identifies pipes by a file descriptor. Pipes are conduits through which data "flow" in one direction. For two-way communication between two tasks, two pipes are needed.

Streams are an abstraction for accessing data. Two types of streams are character streams and binary (integer) streams. Smalltalk has classes which represent internal streams and external streams. Stream classes assume that they are accessing an indexable **Collection,** such as a **String.** The external streams serve as interfaces to data in the operating system. **ExternalStream** establishes protocol for accessing nonhomogeneous collections of data.

Discussion

There is a fixed number of open file descriptors allowed by the operating system. To avoid running out of file descriptors, pipes should always be closed when they are no longer needed. The class variable **OpenPipeStreams** manages the file descriptors held by **PipeStreams.** All open **PipeStreams** can be closed with the message **releaseExternalReferences.**

Snapshots

If you make a snapshot and continue execution, any open **PipeStreams** will still be open. Reloading a snapshot causes all the **PipeStreams** in the class variable **OpenPipeStreams** to be closed. Any subtasks that were running when a snapshot was made will no longer be running when the snapshot image is reloaded. In fact, any communication between an application and the operating system via a **PipeStream** will be terminated when a saved image is reloaded. The operating system closes file descriptors and terminates child tasks when the Smalltalk task terminates.

Class Protocol

Class initialization creates a new collection to record **OpenPipeStreams.**

External References answers the count of open **PipeStreams** and provides a message to close all **PipeStreams.**

There are two *instance creation* methods — one takes an instance of **Pipe** as an argument, the other takes as an argument a file descriptor which is assumed to refer to an operating system pipe.

Instance Protocol

The *initialize-release* method is called by the *instance creation* methods. It sets two instance variables and adds the new **PipeStream** to the class variable **OpenPipeStreams.**

Methods under *accessing* set the instance variable **mode** to either #binary or #text; set or return the instance variable **fileDescriptor**; return the class, **String** or **ByteArray**, used as the buffer for this **PipeStream**; and return the size of the buffer. The message **contents**, inherited from **ReadWriteStream**, is intercepted with "self shouldNotImplement" in **PipeStream — PipeReadStream** provides the method **contentsOfEntireFile**, which is appropriate for that class.

Copying and *positioning* methods are intercepted.

The message **close** disassociates the **PipeStream** from the operating system pipe and removes it from the **OpenPipeStreams** collection. Note that the **PipeStream** may still be referenced by other objects, but typical accessing protocol for it will fail.

Testing protocol answers whether the mode is binary or text, whether **fileDescriptor** references an open operating system pipe, and whether the **PipeStream** is at the end of its data.

Related Classes

Subclasses:

PipeReadStream
PipeWriteStream

You may also want to look at **Pipe**.

```
PipeStream subclass: #PipeWriteStream

    instanceVariableNames:      ''
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Streams'
```

Summary

PipeWriteStreams are used for writing to an operating system pipe. Data are not buffered to the pipe.

Instance Methods

initialize-release

initializeOn: aPipe
Initialize this PipeStream with aPipe, which must contain an open write file descriptor.

accessing

next
Not appropriate for a PipeWriteStream.

next: anInteger
Not appropriate for a PipeWriteStream.

next: anInteger **into:** aCollection
Not appropriate for a PipeWriteStream.

nextPut: aCharacterOrByte
Place aCharacterOrByte into the pipe. Do not buffer the data. Answer the data written.

nextPutAll: aCollection
Write the contents of the collection to the pipe. Don't buffer data into the pipe.

peek
Not appropriate for a PipeWriteStream.

peekFor: anObject
Not appropriate for a PipeWriteStream.

Rationale

PipeWriteStream allows data to be passed to another operating system task.

Discussion

The abstraction of a pipe stream is used outside of Smalltalk, for example, several C functions use a stream for writing, including fprintf and putc. Most users are familiar with the standard output stream, stdout.

Instance creation messages from the superclass should be sent to **PipeWriteStream** to create an instance. Most of the inherited class protocol is not very practical for this class.

The *initialize-release* method initializes appropriate instance variables and adds this **PipeWriteStream** to the class variable **OpenPipeStreams**.

Accessing methods intercept inappropriate inherited messages and implement **nextPut:** and **nextPutAll:** to write a single byte/character or a collection to the stream.

Examples

The following code can be executed in a workspace.

```
stderr ← PipeWriteStream openOnFd: 2.
stderr nextPutAll: 'This gets printed on the screen.'
```

The example opens a **PipeWriteStream** on the file descriptor (2) associated with standard error output, then writes a message on the standard error stream. This might be useful for debug messages while modifying the **Transcript**.

Related Classes

- **OSFilter** uses this class.
- The **PipeStream** hierarchy contains protocol available to **PipeWriteStreams**.
- **Pipe** may also be of interest to you.

```
ExternalPointerData subclass: #PointerArray

    instanceVariableNames:    ''
    classVariableNames:       ''
    poolDictionaries:         ''
    category:                 'OS-Parameters'
```

Summary

Instances of **PointerArray** represent an array of integers or pointers to objects. Elements of the array may be instances of concrete subclasses of **ExternalData** (or other variableByte or variableWord classes with no instance variables) or **Integer.** Every **ExternalData** object is given a pointer to it by the interpreter when certain primitives are invoked. Integers (**SmallIntegers** or **LargeIntegers** up to four bytes long) are given the value of the integer in question, not a pointer to the integer. Since **Integers** are thus treated differently than other objects, the **ExternalData** concrete subclass **IntegerPointer** is used to create a pointer to an integer.

Protocol exists for instance creation, accessing array elements, initializing the inherited **pointers** instance variable, and printing the array.

Instance Methods

initialize-release

initialize
Store the proper offsets into the pointers array.

accessing

at: anIndex
Return the object at anIndex in the pointers area.

at: anIndex **put:** anObject
Place anObject at anIndex in the pointers area.

printing

printOn: outStream
Print a representation of the objects on the receiver.

Class Methods

instance creation

new
Return an empty instance.

new: count
Create an instance with enough space for count data objects.

Rationale

PointerArray is used by the system call class for your operating system to pass arguments to system calls.

Discussion

In the current release (TB2.2.1), **PointerArrays** are only used for arrays of pointers to strings. The **pointers** instance variable of a **PointerArray** has a null-terminated **String** as the object half of each object-offset pair. The interpreter uses **pointers** to fill in the addresses in the space reserved for them in **dataArea**.

Examples

The following code spawns a child task to run the UTek *ls* utility with two arguments, -l and /usr/lib/smalltalk. You could execute the code in a workspace with "printIt" to see the long listing of the contents of the directory /usr/lib/smalltalk.

```
OS executeUtility: '/bin/ls' withArguments: (OrderedCollection
with: '-l' with: '/usr/lib/smalltalk').
```

From the shell, the command line would be `ls -l /usr/lib/smalltalk`. The example accomplishes its work by making an *execve(2)* system call. The *execve(2)* system call is one that expects an array of pointers to strings (actually, pointers to chars) as an argument. The following code (extracted from non-portable code found in the **UTekSystemCall execute:withArguments:withEnvironment:** method) constructs the **PointerArray** needed by the spawned task given in the example above. ArgCollection is an **OrderedCollection** specified in **executeUtility:withArguments:**. ArgCollection has two elements: '-l' and '/usr/lib/smalltalk'.

```
args ← PointerArray new: argCollection size + 1. "nil at end"
i ← 1.
argCollection do:
    [:arg| args at: i put: arg, StringTerminator.
```

```
i ← i + 1].
```

An instance of **PointerArray** is created, one element larger than the number of arguments — the extra element (nil) is required by the operating system to mark the end of the array. The temporary variable, *i*, is the index into the **PointerArray**. The **PointerArray** is filled by copying the elements from argCollection with a **StringTerminator** appended to each element.

Related Classes

UTekSystemCall uses this class.

```
ExternalBinaryData variableByteSubclass: #Rlimit

    instanceVariableNames:      ''
    classVariableNames:         'CurDataIndex MaxDataIndex '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

Rlimit provides creation and accessing protocol for the following C structure.

```
struct rlimit {
    long    rlim_cur;    /* current (soft) limit */
    long    rlim_max;    /* maximum value for rlim_cur */
}
```

The structure is documented under *getrlimit(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

CurDataIndex

MaxDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **CurDataIndex** holds the offset of the "rlim_cur" field.

Instance Methods

accessing

cur Return the value of the structure field named cur.

cur: anInt
Assign the argument, anInt, to the structure field named cur.

cur: anInt **max:** anotherInt
Assign values to all the fields of the structure.

max
Return the value of the structure field named max.

max: anInt
Assign the argument, anInt, to the structure field named max.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

cur: anInt **max:** anotherInt
Return an instance with the values of the fields assigned.

Rationale

The structure is used in support of the following UTek system calls:

getrlimit(2)
setrlimit(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
ExternalBinaryData variableByteSubclass: #Rusage

    instanceVariableNames:     ''
    classVariableNames:        'IdrssDataIndex InblockDataIndex IsrssDataIndex
                               IxrssDataIndex MajfltDataIndex MaxrssDataIndex
                               MinfltDataIndex MsgrcvDataIndex
                               MsgsndDataIndex NivcswDataIndex
                               NsignalsDataIndex NswapDataIndex
                               NvcswDataIndex OublockDataIndex
                               StimeSecDataIndex StimeUsecDataIndex
                               UtimeSecDataIndex UtimeUsecDataIndex '
    poolDictionaries:          ''
    category:                  'OS-Parameters'
```

Summary

Rusage provides creation and accessing protocol for the following C structure.

```
struct rusage {
    struct timeval  ru_utime;     /* user time used */
    struct timeval  ru_stime;     /* system time used */
    long            ru_maxrss;
    long            ru_ixrss;     /* integral shared memory size */
    long            ru_idrss;     /* integral unshared data size */
    long            ru_isrss;     /* integral unshared stack size */
    long            ru_minflt;    /* page reclaims */
    long            ru_majflt;    /* page faults */
    long            ru_nswap;     /* swaps */
    long            ru_inblock;   /* block input operations */
    long            ru_oublock;   /* block output operations */
    long            ru_msgsnd;    /* messages sent */
    long            ru_msgrcv;    /* messages received */
    long            ru_nsignals;  /* signals received */
    long            ru_nvcsw;     /* voluntary context switches */
    long            ru_nivcsw;    /* involuntary context switches */
}
```

The structure is documented under *getrusage(2)* in the manual *UTek Command Reference, Volume 2*.

Class Variables

IdrssDataIndex

InblockDataIndex

IsrssDataIndex

IxrssDataIndex

MajfltDataIndex

MaxrssDataIndex

MinfltDataIndex

MsgrcvDataIndex

MsgsndDataIndex

NivcswDataIndex

NsignalsDataIndex

NswapDataIndex

NvcswDataIndex

OublockDataIndex

StimeSecDataIndex

StimeUsecDataIndex

UtimeSecDataIndex

UtimeUsecDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **IdrssDataIndex** holds the offset of the "ru_idrss" field.

When a field in a structure is another C structure, separate class variables are created for each field in the other structure. For example, the class variable **UtimeSecDataIndex** holds the offset of the "sec" field of the "ru_utime" structure, timeval.

Instance Methods

accessing

idrss
Return the value of the structure field named idrss.

inblock
Return the value of the structure field named inblock.

isrss
Return the value of the structure field named isrss.

ixrss
Return the value of the structure field named ixrss.

majflt
Return the value of the structure field named majflt.

maxrss
Return the value of the structure field named maxrss.

minflt
Return the value of the structure field named minflt.

msgrcv
Return the value of the structure field named msgrcv.

msgsnd
Return the value of the structure field named msgsnd.

nivcsw
Return the value of the structure field named nivcsw.

nsignals
Return the value of the structure field named nsignals.

nswap
Return the value of the structure field named nswap.

nvcsw
Return the value of the structure field named nvcsw.

oublock
Return the value of the structure field named oublock.

stime
Return the value of the structure field named stime.

stimeSec
Return the sec field of the structure at the field named stime.

stimeUsec
Return the usec field of the structure at the field named stime.

utime
Return the value of the structure field named utime.

utimeSec
Return the sec field of the structure at the field named utime.

utimeUsec
Return the usec field of the structure at the field named utime.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

callingProcess
Return the rusage structure for the calling process.

terminatedChildProcesses
Return the rusage structure for all terminated child processes of the current process.

Rationale

The rusage C structure is used in support of the following UTek system calls:

getrusage(2)
wait3(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
View subclass: #ScreenView

    instanceVariableNames:      ''
    classVariableNames:         'DefaultModel '
    poolDictionaries:           ''
    category:                   'Interface-Support'
```

Summary

ScreenView is the view for the parts of the display screen that have no window on them. It responds to **restore:**, so that when views erase themselves, the background gets restored. Its model must respond to **displayOn:fill:**. An **InfiniteForm** whose **patternForm** is 16x16 (a halftone) makes a good model, because such a form quickly refreshes the screen. The default color is gray.

The following message can be used as a sample to change the model of the current screenController. It will require you to provide a form by framing a section of the screen.

```
ScreenController backgroundFromUser.
```

Class Variables

DefaultModel <InfiniteForm>

The default model for new screen views. New screen views are created upon opening a project.

Instance Methods

initialize-release

initialize

Specifies the default model for this view. Its controller is ScreenController.

controller access

defaultControllerClass

Answer the class of ScreenView's controller.

displaying

displayView
Paint the Display using the ScreenView's model.

restore: rectangleCollection
Redisplay those portions of the receiver that intersect rectangles within rectangleCollection. Answer an empty collection of rectangles, since the screen is everywhere.

Class Methods

examples

backgroundFromUser
Frame a small portion of the display and it will become the model for the background.

darkGrayBackground
Change the background to dark gray.

grayBackground
Change the background form to gray.

whiteBackground
Change the background form to white.

model accessing

defaultModel
Answer the default background model. If it is nil, set it to a grayMask (a 16x16 form).

defaultModel: aForm
Change the background form to aForm.

Rationale

This class allows you to change the background on the screen to any pattern or mask that you prefer. Individual projects can have their own unique background.

Discussion

Initialize-release protocol initializes the default model and the controller for a **ScreenView**.

Controller access protocol answers the controller for this class.

Displaying protocol paints and restores the display.

Examples contains several methods to change the background color (to white, gray, or dark gray) and one method that requires you to frame a small section of the display to become the model for the background.

Model accessing protocol allows you to set the default model for the background to any form you choose. It also answers the default background or sets it to gray if it isn't already set.

Examples

See the methods in the metaclass *examples* message category for ideas on how to use this class.

Related Classes

Form
InfiniteForm
ScreenController

```
ExternalBinaryData variableByteSubclass: #Sgttyb

    instanceVariableNames:    ''
    classVariableNames:       'EraseDataIndex FlagsDataIndex
                              IspeedDataIndex KillDataIndex
                              OspeedDataIndex '
    poolDictionaries:         ''
    category:                 'OS-Parameters'
```

Summary

Sgttyb provides creation and accessing protocol for the following C structure.

```
struct sgttyb {
    char    sg_ispeed;    /* input speed */
    char    sg_ospeed;    /* output speed */
    char    sg_erase;     /* erase character */
    char    sg_kill;      /* kill character */
    short   sg_flags;     /* mode flags */
}
```

The structure is documented under *tty(4)* in the manual *UTek Command Reference, Volume 2*.

Class Variables

EraseDataIndex

FlagsDataIndex

IspeedDataIndex

KillDataIndex

OspeedDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **EraseDataIndex** holds the offset of the "sg_erase" field.

Instance Methods

accessing

erase
Return the value of the structure field named erase.

erase: aCharacter
Assign the argument, aCharacter, to the structure field named erase.

flags
Return the value of the structure field named flags.

flags: anInt
Assign the argument, anInt, to the structure field named flags.

ispeed
Return the value of the structure field named ispeed.

ispeed: anInt
Assign the argument, anInt, to the structure field named ispeed.

ispeed: anInt **ospeed:** anotherInt **erase:** eCharacter
kill: kCharacter **flags:** lastInt
Assign values to all the fields of the structure.

kill Return the value of the structure field named kill.

kill: aCharacter
Assign the argument, aCharacter, to the structure field named kill.

ospeed
Return the value of the structure field named ospeed.

ospeed: anInt
Assign the argument, anInt, to the structure field named ospeed.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize

Assign offset values to the class variables and define the size of the structure.

instance creation

default

Return an instance containing the default characters.

Rationale

The structure is used in support of the following UTek system call:

ioctl(2)

Related Classes

UTekSystemCall implements the system call listed above.

```
ExternalBinaryData variableByteSubclass: #SockaddrIn

    instanceVariableNames:    ''
    classVariableNames:       'AddrDataIndex FamilyDataIndex PortDataIndex
                              ZeroDataIndex ZeroLength '
    poolDictionaries:         ''
    category:                 'OS-Parameters'
```

Summary

SockaddrIn provides creation and accessing protocol for the following C structure.

```
struct sockaddr_in {
    short           sin_family;
    u_short         sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
}
```

The structure is documented under *inet(4N)* in the manual *UTek Command Reference, Volume 2*. It is used as an internet domain socket address.

Class Variables

AddrDataIndex

FamilyDataIndex

PortDataIndex

ZeroDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **PortDataIndex** holds the offset of the "sin_port" field.

ZeroLength <Integer>
This variable holds the constant, 8, of the sin_zero field.

Instance Methods

accessing

addr
Return the value of the structure field named addr.

addr: anInternetAddress
Assign the argument, anInternetAddress, to the structure field named addr.

port
Return the value of the structure field named port.

port: anInt
Assign the argument, anInt, to the structure field named port.

port: anInt **addr:** anInternetAddress
Assign values to all the fields of the structure.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

port: anInt **addr:** anInternetAddress
Return an instance with the values of the fields assigned.

Rationale

This class is the C structure which holds an internet domain socket address. It holds the network address and the port number, for accessing processes such as *ftp*, *telnet*, and *finger*. The structure is used in support of the following UTek system calls:

accept(2)
bind(2)
connect(2)
getpeername(2)
getsockname(2)
recvfrom(2)
sendto(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
ExternalBinaryData variableByteSubclass: #SockaddrUn

    instanceVariableNames:      ''
    classVariableNames:         'FamilyDataIndex PathDataIndex PathLength '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

SockaddrUn provides creation and accessing protocol for the following C structure.

```
struct sockaddr_un {
    short   sun_family;      /* AF_UNIX */
    char    sun_path[108];   /* path name */
}
```

The structure is found in *sys/un.h*. It is used as a UNIX domain socket address.

Class Variables

FamilyDataIndex

PathDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **PathDataIndex** holds the offset of the "sun_path" field.

PathLength <Integer>
This variable holds the constant, 108, of the sun_path field.

Instance Methods

accessing

family: anInt **path:** aByteArray
Assign values to all the fields of the structure.

path
Return the value of the structure field named path.

path: aString
Assign values to all the fields of the structure.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

family: anInt **path:** aByteArray
Return an instance with the values of the fields assigned.

path: aString
Return an instance with the values of the fields assigned.

Rationale

This class represents the C structure which holds a UNIX domain socket address. It allows access to unrelated processes running on the local machine. The structure is used in support of the following UTek system calls:

accept(2)
bind(2)
connect(2)
getpeername(2)
getsockname(2)
recvfrom(2)
sendto(2)

Related Classes

UTekSystemCall implements the system calls listed above.

```
ExternalBinaryData variableByteSubclass: #Stat

    instanceVariableNames:     ''
    classVariableNames:        'AtimeDataIndex BlksizeDataIndex
                               BlocksDataIndex CtimeDataIndex DevDataIndex
                               GidDataIndex HostidDataIndex InoDataIndex
                               ModeDataIndex MtimeDataIndex NlinkDataIndex
                               RdevDataIndex SizeDataIndex Spare1DataIndex
                               Spare2DataIndex Spare3DataIndex
                               Spare4DataIndex UidDataIndex '
    poolDictionaries:          ''
    category:                  'OS-Parameters'
```

Summary

Stat provides accessing protocol for the following C structure.

```
struct stat {
    dev_t           st_dev;      /* ID of device containing a directory entry
                                    for this file */
    ino_t           st_ino;      /* this inode's number */
    unsigned short  st_mode;     /* file mode */
    short           st_nlink;    /* number of hard links to the file */
    short           st_uid;      /* user ID of the file's owner */
    short           st_gid;      /* group ID of the file's group */
    dev_t           st_rdev;     /* ID of device — this entry is defined only
                                    for character special or block special files */
    off_t           st_size;     /* total size of file */
    time_t          st_atime;    /* time of last access */
    int             st_spare1;
    time_t          st_mtime;    /* time of last data modification */
    int             st_spare2;
    time_t          st_ctime;    /* time of last file status change */
    int             st_spare3;
    long            st_blksize;  /* optimal blocksize for file system I/O ops */
    long            st_blocks;   /* actual number of blocks allocated */
    long            st_hostid;   /* hostid of machine where file is located */
    long            st_spare4;
}
```

The structure is documented under *stat(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

AtimeDataIndex

BlksizeDataIndex

BlocksDataIndex

CtimeDataIndex

DevDataIndex

GidDataIndex

HostidDataIndex

InoDataIndex

ModeDataIndex

MtimeDataIndex

NlinkDataIndex

RdevDataIndex

SizeDataIndex

Spare1DataIndex

Spare2DataIndex

Spare3DataIndex

Spare4DataIndex

UidDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **AtimeDataIndex** holds the offset of the "st_atime" field.

Instance Methods

accessing

atime
Return the value of the structure field named atime.

blksize
Return the value of the structure field named blksize.

blocks
Return the value of the structure field named blocks.

ctime
Return the value of the structure field named ctime.

dev
Return the value of the structure field named dev.

gid Return the value of the structure field named gid.

hostid
Return the value of the structure field named hostid.

ino Return the value of the structure field named ino.

mode
Return the value of the structure field named mode.

mtime
Return the value of the structure field named mtime.

nlink
Return the value of the structure field named nlink.

rdev
Return the value of the structure field named rdev.

size
Return the value of the structure field named size.

spare1
Return the value of the structure field named spare1.

spare2
Return the value of the structure field named spare2.

spare3
Return the value of the structure field named spare3.

spare4
Return the value of the structure field named spare4.

uid Return the value of the structure field named uid.

accessing-status

ifdir
If the receiver is a status for a directory, return true.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

Rationale

The structure is used in support of the following UTek system calls:

fstat(2)
lstat(2)
stat(2)

Related Classes

UTekSystemCall implements the system calls listed above.

StrikeFont

Object subclass: #StrikeFont

instanceVariableNames:	'xTable glyphs name stopConditions type minAscii maxAscii maxWidth strikeLength ascent descent raster subscript superscript emphasis ascentForStdAsciiChars descentForStdAsciiChars '
classVariableNames:	'ASCIICompatible DefaultStopConditions DefaultStopConditionsForMonospaceFonts FaceNames '
poolDictionaries:	'TextConstants '
category:	'Graphics-Support'

Summary

StrikeFonts are a compact encoding of a set of **Forms** corresponding to characters in the ASCII character set. Additional characters, other than standard ASCII, may be present depending upon the type of font. All the forms are placed side by side in a large form whose height is the font height, and whose width is the sum of all the character widths. The **xTable** gives the left x-coordinates of the sub-forms corresponding to the characters.

Instance Variables

ascent <Integer>

Maximum extent of characters above the baseline.

ascentForStdAsciiChars <Integer>

Maximum extent of characters above the baseline for ASCII decimal equivalent 32 through 126.

descent <Integer>

Maximum extent of characters below the baseline.

descentForStdAsciiChars <Integer>

Maximum extent of characters below the baseline for ASCII decimal equivalent 32 through 126.

emphasis <Integer>

This code indicates that the face is achieved synthetically by altering another face: 0=none, 1=bold, 2=italic, 4=underline, 8=strike-out, 16=subscript, and 32=superscript.

glyphs <Form>

An instance of Form containing bits representing the entire set of characters in this font.

maxAscii <Integer>

Highest ASCII value supported by this font.

maxWidth <Integer>

Width of widest character. Not presently used, but may be for font modification.

minAscii <Integer>

Lowest ASCII value supported by this font.

name <String>

Name of this font.

raster <Integer>

Actual width of glyphs (strikeLength + 15 \\ 16), given Forms are padded to multiples of 16 bits.

stopConditions <Array>

Array at least as large as xTable with an entry for each character in the font. Nil indicates no special processing; any other entry causes special processing to be executed for the character (e.g., tab, linefeed) by the CompositionScanner.

strikeLength <Integer>

Width of glyphs.

subscript <Integer>

Additional vertical offset relative to the baseline.

superscript <Integer>

Additional vertical offset relative to the baseline.

type <Integer>

Code indicating font type:

Code	Font
1	Tektronix monospaced
2	Tektronix proportional
3	Xerox
n	A font from other sources

xTable <Array>

Left x-coordinate of character sub-forms in glyphs. The xTable entry of a character is the answer from aCharacter asciiValue plus one. For example,

the left x-coordinate of $A is the 66th entry in the xTable of the default font in the default text style (in the standard image).

Class Variables

ASCIICompatible <Integer>

Constant used when the font is loaded to determine whether it is ASCII compatible (0 = unknown type, 1 = ASCII).

DefaultStopConditions <Array>

DefaultStopConditions is a class variable containing a stop condition entry for each character in a standard ASCII proportional font (type 2 or 3). Unless another stopCondition array is initialized when the StrikeFont is loaded, a newly created standard proportional StrikeFont's stopCondition array will refer to DefaultStopConditions.

DefaultStopConditionsForMonospaceFonts <Array>

DefaultStopConditionsForMonospaceFonts is a class variable containing a stop condition entry for each character in a monospace font (type = 1). Unless another stopCondition array is initialized when the StrikeFont is loaded, a newly created standard monospace StrikeFont's stopCondition array will refer to DefaultStopConditionsForMonospaceFonts.

FaceNames <Dictionary>

Dictionary of valid face names (e.g., 'Bold', 'Bold Italic') and associated font name sub-strings (e.g., 'B', 'X').

Pool Dictionaries

TextConstants

A dictionary of symbols for non-printing characters, symbols related to text composition and text emphasis, and default values for text composition and text emphasis.

Instance Methods

initialize-release

initializeFrom: aFontFile

Read a font from aFontFile which must be in Tektronix format. Return a StrikeFont or, in case of error, nil.

accessing

ascent
Answer the font's maximum extent of characters above the baseline.

ascentForStdAsciiChars
Answer the font's maximum extent of characters above the baseline for ASCII decimal equivalent 32 through 126.

bottomLead: character
Answer the amount of white space or leading imbedded in the character form at the bottom.

characterForm: character
Answer a Form copied out of the glyphs for this character.

descent
Answer the font's maximum extent of characters below the baseline.

descentForStdAsciiChars
Answer the font's maximum extent of characters below the baseline for ASCII decimal equivalent 32 through 126.

glyphs
Answer a Form containing the bits representing the characters of the receiver.

height
Answer the height of the font — the total of maximum extents of characters above and below the baseline.

leadinfo
If this is a fixed pitch font, compute the leading by adding the font's imbedded top and bottom leading. If this is a proportional font, return a recommended leading adjusted according to the height of the font. If the font size is not standard, return a nominal leading.

maxAscii
Answer the integer that is the last ASCII character value of the receiver.

maxWidth
Answer the integer that is the width of the receiver's widest character.

minAscii
Answer the integer that is the first ASCII character value of the receiver.

name
Answer the receiver's name.

name: aString
Set the receiver's name.

raster
Answer an integer that specifies the layout of the glyphs' form.

stopConditions
Answer the array of selectors to be performed in scanning text made up of the receiver's characters.

subscript
Answer an integer that is the additional vertical offset relative to the baseline for positioning characters as subscripts.

subscript: anInteger
Set the subscript instance variable that is the additional vertical offset relative to the baseline for positioning characters as subscripts.

superscript
Answer an integer that is the additional vertical offset relative to the baseline for positioning characters as superscripts.

superscript: anInteger
Set the superscript instance variable that is the additional vertical offset relative to the baseline for positioning characters as superscripts.

tightLeadinfo
If this is a fixed pitch font, compute the leading by adding the font's imbedded top and bottom leading. If this is a proportional font, return a minimum recommended leading of 4.

topLead: character
Answer the amount of white space or leading imbedded at the top of the character form.

type
Answer the receiver's compatibility mode.

type: anInteger
Set the receiver's compatibility mode.

unprintableCharacter
Return a character that represents all unprintable characters in the font.

widthOf: aCharacter
Answer the width of the argument aCharacter.

xTable
Answer an array of the left x-coordinates of characters in glyphs.

converting

asTextStyle
Return a TextStyle composed of the StrikeFont which received this message.

displaying

characters: anInterval **in:** sourceString **displayAt:** aPoint **clippedBy:** clippingRectangle **rule:** ruleInteger **mask:** aForm
Simple, slow method for displaying a line of characters. No wrap-around is handled.

composeWord: aTextLineInterval **in:** sourceString **beginningAt:** xInteger
Return the sum of the widths of characters in sourceString starting at xInteger for aTextLineInterval count. This method is similar to performance of the scanning primitive, but ignores stop conditions.

displayLine: aString **at:** aPoint
Display the characters in aString, starting at position aPoint.

emphasis

emphasis
Answer the integer code for one of the following: synthetic bold, italic, underline, or strike-out. If the font has no synthetic emphasis, a zero value will be returned.

emphasis: code
Set the emphasis code to synthesize one of these: bold=1, italic=2, underlined=4, struck out=8, subscript=16, superscript=32. Zero means no synthetic emphasis.

emphasized: code
Answer a copy of the receiver, and set the returned StrikeFont's emphasis to code.

emphasized: code **named:** aString
Answer a copy of the receiver, with emphasis set to code and name set to aString.

printing

printOn: aStream
Print the name and emphasis of this font on aStream.

writeOn: aStream
Write out the representation of this font on aStream using the Tektronix font file format.

writeOnFile: aString
Create a Tektronix font format file named aString. Types 1 (monospace), 2 (proportional) and 3 (Xerox) can be reread with an instance creation message without a warning notice in the System Transcript.

testing

checkCharacter: character
Return a character if it is a valid printable character in the receiver; otherwise return the default character for unprintable characters.

isFixedPitch
Answer true if all legal characters of the font are the same width.

isVirtual
Since the receiver is not a VirtualStrikeFont, return false.

Class Methods

class initialization

initialize
Set up FaceNames dictionary and the default stop conditions.

fileIn-Out

readAll: aFontDirectoryName
Read in all the fonts in aFontDirectoryName. These fonts are assumed to be in Tektronix format. Skip over files with illegal names, and files that return 'nil' when read.

readFrom: aFontFileStream
Read in a font from aFontFileStream. This font is assumed to be in Tektronix format.

Rationale

StrikeFont enables characters to display, so that when you type on the keyboard, graphic representations of the ASCII codes will echo to the display. Since Smalltalk is graphically oriented, instances of **StrikeFont** allow you to choose among collections of characters with different physical appearances.

Background

Terminology

Before the Smalltalk implementation of fonts is discussed, you might find it helpful to understand some of the terms used in the descriptions of the *Graphics — Support* classes in this manual.

Face is the emphatic property of a font. For Tektronix and Xerox fonts, the last one or two characters in the **StrikeFont** name will signify the face, as discussed later under "StrikeFont Names". Face is represented in **StrikeFont** by the **emphasis** instance variable when it is necessary to synthesize a particular face. Basal is the "base" or standard face from which other fonts are synthesized. Synthesizing is discussed below under "Available Fonts".

Family refers to the basic look of a set of characters that makes it distinguishable from another set. Family is the intrinsic property of a font. Families are named and frequently protected by copyright. Examples include "Helvetica" and "Pellucida". The array of fonts which are returned by the message to the global **TextStyleManager**

```
StyleManager styleName:'Pellucida Serif 10-12' baseNames:
#('PellucidaSerif10' 'PellucidaSerif12')
```

will all belong to the "Pellucida" family. Usually, the fonts in a **TextStyle** will all be in one family, although you may mix families in a **TextStyle.** This manual was produced using fonts in three families, "Helvetica", "TimesRoman", and "Courier".

A *font*, in the days before computer typesetting, was a set of letters and symbols, such as punctuation, that a printer would assemble in lines, coat with ink, and press — to put impressions on paper. In Smalltalk, a font is an instance of **StrikeFont**, a collection of **Forms** with the bitmaps for each character and symbol available in the font.

The *size* of a font is measured vertically in *points* in the world of printing. A point is 1/72 of an inch. For historical reasons, the point size has been retained in the font names in Smalltalk, although the size on the display will be an approximation of point size. To draw a parallel to a Smalltalk instance variable, the **ascent** of a **StrikeFont** is close to the "size" in the corresponding font file's name.

Leading is the sum, in points, of the font size and the white space below a line of printed text. The scale of leading in Smalltalk is pixels, not points. In printing the spoken phrase "10 on 12" would indicate 10 point type on 12 point leading. Smalltalk's parallel to leading is the **lineGrid** in a **TextStyle**. Several **StrikeFont** *accessing* methods answer the amount of white space imbedded at the top or bottom of a character, answer the recommended leading for a font, or the minimum leading for a font. Fixed pitch fonts in this product have "leading" imbedded at the top and bottom of the forms in **glyphs**, so no extra leading is recommended when you create a **TextStyle** with the "Pellucida Typewriter" family.

Font Files

The Smalltalk fonts are derived from font files in the Utek operating system residing in the directory name returned by

```
OS fontDirectory fullName.
```

This directory contains quite a number of font files. The files having Pellucida and Xerox as part of their names are completely compatible with Tektronix Smalltalk. By compatible is meant that the files are in Tektronix font file format, and the methods for installing fonts will work properly. Other fonts in this directory may be loaded in the **FontManager**, but they may have a character set that differs from familar ones.

The font files are grouped into font families: Pellucida Sans-Serif, Pellucida Serif, Pellucida Typewriter, Xerox Sans-Serif, and Xerox Serif. The Pellucida Sans-Serif, Pellucida Serif, and Xerox families are proportionally spaced (individual characters within the same font have varying widths); the Pellucida Typewriter family is monospaced (individual characters within the same font have the same width). The Pellucida families have been specially designed for Tektronix Smalltalk; the Xerox families are the standard Smalltalk-80 Version 2 fonts.

Reading and Writing Font Files

StrikeFont has methods for reading and writing Tektronix font files. Note that whenever Smalltalk reads a Tektronix font file, it switches the character position of the up arrow character (↑) and left arrow (←) with the caret (^) and underscore (_) characters. If you ask the character ↑, for instance, what its **asciiValue** is you get 94.

The method to write a **StrikeFont** takes care to switch the positions of the ↑, ←, ^, and _ characters if the type of the **StrikeFont** is either 1 (Tektronix monospaced) or 2 (Tektronix proportionally spaced). This ensures that the proportional or monospaced fonts written by Smalltalk have consistent ordering for standard ASCII characters.

Available Fonts

In some cases a face other than Basal is created for an instance of **StrikeFont** by synthesizing it from another face. Synthesizing involves bitmap manipulation, for example, shearing Basal to create Italic, copying and offsetting Basal to create Bold. The underlined fonts are synthesized from their corresponding font, for example, Bold Underlined is synthesized from Bold. Non-synthetic faces usually have a better appearance than synthetic faces.

There are 64 non-synthetic Pellucida fonts. The Pellucida sans-serif and serif fonts are available in four non-synthetic faces (Basal, Bold, Italic, and BoldItalic) and seven sizes (8, 10, 12, 14, 18, 24, and 36 point). The Pellucida Typewriter fonts are available in two non-synthetic faces (Basal and Bold) and four sizes (10, 12, 16, and 18 point).

There are 11 non-synthetic Xerox fonts. The serif and sans-serif fonts are available in 10 and 12 point size and Basal, Bold, and Italic. The sans-serif 10 point Italic, however, is synthesized.

Font File Names

Font file names are made up of four parts:

- A name descriptive of the family the font belongs to. Examples are *PellucidaSans-Serif*, *PellucidaSerif*, *PellucidaTypewriter*, and *XeroxSans-Serif*.
- The point size of the font. This is usually an even number from *8* to *36*.
- Optionally, one of the following emphasis characters: *B* standing for bold, *I* for italic, and *X* for bold italic. Absence of one of those three characters denotes basal.

- The required suffix *font*, denoting a font file.

Note that some font files are not entirely compatible (a clue is that they do not have Pellucida or Xerox in their name). Examples are *BertrandVariable12.font*, *MagnoliaFixed12.font*, and *Micro5.font*. The primary incompatibility of these files is that underscore is not mapped to the Smalltalk assignment (left) arrow and the caret (shift-6) is not mapped to the Smalltalk return (up) arrow. In general, even fonts with compatibility code other than 1, 2, or 3 can be loaded as **StrikeFonts**.

StrikeFont Names

The names of **StrikeFonts** are closely related to font file names, however, **StrikeFont** names are not constructed from font file names themselves, but from information *in* a font file.

The name of a **StrikeFont** is a **String** with three components (family, size, and face) and no imbedded spaces. The family component is the family name with spaces removed; the size component is the **printString** of the numeric size; and the face component is a **String** of length zero, one, or two signifying the face. The supported face codes are "" (Basal), **B** (Bold), **I** (Italic), **X** (BoldItalic), **U** (Basal Underlined), **BU** (Bold Underlined), **IU** (Italic Underlined), and **XU** (BoldItalic Underlined). The face codes for synthetic fonts are not taken from the font file; for example, the **U** for "underlined" does not appear in the font file. Examples of names include 'PellucidaSans-Serif8', 'XeroxSerif12I', and 'PellucidaTypewriter18BI'. The underlined face codes are assigned when a **styleName:baseNames:** message is sent to the global **StyleManager** to create an instance of **TextStyle**.

Discussion

Class Protocol

Class initialization contains one method to initialize the class; it is not a message a user would typically send.

FileIn-Out methods file in **StrikeFonts** from the fonts directory on the disk — either one font or all the fonts in a specified directory. The recommended way to install fonts is to use a **TextStyleManager** accessing message.

Instance Protocol

You are not likely to use the *initialize-release* method **initializeFrom:**. It is called by the class method **readFrom:**.

Accessing methods return the values of instance variables. One method, **characterForm:**, returns a form with a sub-form from **glyphs**; you could use the bit editor to modify the form or simply display the single character. You can set the **name** and **type** of a **StrikeFont**, and the amount to subscript and superscript the characters. You shouldn't change the **type** of a **StrikeFont** unless you have modified it, for example, by adding or removing characters.

A *converting* method converts a **StrikeFont** to a **TextStyle** and returns the **TextStyle**.

Displaying methods enable you to compose and display characters without including them in typical displayable objects such as a **Paragraph**. You can determine the width of part of a string, display part or all of a string in a form, or display a string at a point you specify. The two methods for displaying do not wrap the characters at the right boundary of the form or display.

Emphasis methods answer or set the emphasis of the **StrikeFont**, return a copy of the **StrikeFont** with emphasis set to the previous value plus the specified additional amount, and return a copy with the specified additional emphasis and a specified name.

Printing method **printOn:** prints the class name, **name** instance variable, and **emphasis** instance variable of a **StrikeFont** on a stream. Two methods, **writeOn:** and **writeOnFile:**, write the **StrikeFont** on a stream or a disk file, respectively. The font is written in a form which is readable by the method which loads fonts.

Testing methods check whether a character is within the ASCII range of the **StrikeFont** and answer whether all the characters in the **StrikeFont** are the same width (fixed pitch).

Examples

The code below shows one way to insert in a string a character for which there is no key on your keyboard. To see what Character value: 2 is, try executing the code in a workspace.

```
s ← 'This is a test.'.
s at: 15 put: (Character value:2).
s asDisplayText displayAt: Sensor waitClickButton.
```

The tables of characters available in the Pellucida fonts show you the argument to **value:** to create an instance of the character. The argument should be the number in the box with the character you want. (See tables at the end of this class.)

Related Classes

StrikeFontManager
TextStyle
TextStyleManager
VirtualStrikeFont

Tektronix Proportional Fonts
Pellucida Serif and Sans-Serif
(Compatibility Code = 2)
Characters 0 - 127

0	~ 16	space 32	0 48	@ 64	P 80	‘ 96	p 112
ˇ 1	ffi 17	! 33	1 49	A 65	Q 81	a 97	q 113
¿ 2	ffl 18	” 34	2 50	B 66	R 82	b 98	r 114
Ç 3	em dash 19	# 35	3 51	C 67	S 83	c 99	s 115
¨ 4	fi 20	$ 36	4 52	D 68	T 84	d 100	t 116
` 5	fl 21	% 37	5 53	E 69	U 85	e 101	u 117
ff 6	en dash 22	& 38	6 54	F 70	V 86	f 102	v 118
‘ 7	˘ 23	’ 39	7 55	G 71	W 87	g 103	w 119
¡ 8	— 24	(40	8 56	H 72	X 88	h 104	x 120
9	n space 25	) 41	9 57	I 73	Y 89	i 105	y 121
10	∞ 26	* 42	: 58	J 74	Z 90	j 106	z 122
´ 11	↑ 27	+ 43	; 59	K 75	[91	k 107	{ 123
12	← 28	, 44	< 60	L 76	\ 92	l 108	| 124
13	· 29	- 45	= 61	M 77	] 93	m 109	} 125
— 14	~ 30	. 46	> 62	N 78	^ 94	n 110	~ 126
m space 15	° 31	/ 47	? 63	O 79	_ 95	o 112	▮ 127

The number in the lower left corner of each cell is the ASCII decimal value (returned by sending the message **asciiValue** to the Character).

3393-1

Tektronix Monospace Font
Pellucida Typewriter
(Compatibility Code =1)
Characters 0-127

0	16	space 32	0 48	@ 64	P 80	‘ 96	p 112
1	17	! 33	1 49	A 65	Q 81	a 97	q 113
2	18	" 34	2 50	B 66	R 82	b 98	r 114
3	19	# 35	3 51	C 67	S 83	c 99	s 115
4	20	$ 36	4 52	D 68	T 84	d 100	t 116
5	21	% 37	5 53	E 69	U 85	e 101	u 117
6	22	& 38	6 54	F 70	V 86	f 102	v 118
7	23	' 39	7 55	G 71	W 87	g 103	w 119
8	24	(40	8 56	H 72	X 88	h 104	x 120
9	n space 25	) 41	9 57	I 73	Y 89	i 105	y 121
10	↑ 26	* 42	: 58	J 74	Z 90	j 106	z 122
11	← 27	+ 43	; 59	K 75	[91	k 107	{ 123
12	28	, 44	< 60	L 76	\ 92	l 108	| 124
13	29	- 45	= 61	M 77	] 93	m 109	} 125
14	30	. 46	> 62	N 78	^ 94	n 110	~ 126
m space 15	31	/ 47	? 63	O 79	_ 95	o 111	■ 127

The number in the lower left corner of each cell is the ASCII decimal value (returned by sending the message **asciiValue** to the Character).

3393-2

Tektronix Monospace Font

Pellucida Typewriter
(Compatibility Code = 1)
Characters 128-255

NU 128	DL 144	SP 160	0 176	– 192	Ñ 208	◆ 224	240
SH 129	D1 145	Ä 161	1 177	¢ 193	ñ 209	█ 225	241
SX 130	D2 146	ä 162	2 178	¦ 194	¿ 210	HT 226	242
EX 131	D3 147	Å 163	3 179	† 195	i 211	FF 227	243
ET 132	DA 148	å 164	4 180	□ 196	a 212	CR 228	244
EQ 133	NK 149	Æ 165	5 181	■ 197	s 213	LF 229	245
AK 134	SY 150	æ 166	6 182	● 198	t 214	° 230	246
BL 135	EB 151	à 167	7 183	Δ 199	r 215	± 231	247
BS 136	CN 152	Ç 168	8 184	d 200	µ 216	NL 232	248
HT 137	EM 153	é 169	9 185	I 201	Σ 217	VT 233	≤ 249
LF 138	SB 154	è 170	ù 186	202	Ω 218	234	≥ 250
VT 139	EC 155	Ö 171	b 187	203	⌠ 219	235	π 251
FF 140	FS 156	ö 172	⊙ 188	204	⌡ 220	236	≠ 252
CR 141	GS 157	ø 173	¤ 189	205	÷ 221	237	£ 253
SO 142	RS 158	Ü 174	§ 190	¬ 206	≈ 222	238	■ 254
SI 143	US 159	ü 175	¨ 191	∝ 207	┌ 223	239	DT 255

The number in the lower left corner of each cell is the ASCII decimal value (returned by sending the message **asciiValue** to the Character).

3393-3

StrikeFontManager

```
Dictionary variableSubclass: #StrikeFontManager

    instanceVariableNames:      ''
    classVariableNames:         'Emphases '
    poolDictionaries:           'TextConstants '
    category:                   'Graphics-Support'
```

Summary

StrikeFontManager is a **Dictionary** which maps instances of **StrikeFont** and **VirtualStrikeFont** to **String** names.

Class Variables

Emphases <Dictionary>

Contains associations between a valid emphasis string and its corresponding emphasis code.

Pool Dictionaries

TextConstants

A dictionary of symbols for non-printing characters, symbols related to text composition and text emphasis, and default values for text composition and text emphasis.

Instance Methods

accessing

at: aString **put:** aStrikeFont

Install aStrikeFont named aString.

familySizeFace: name

Return an array with name <String>, pointSize <Integer>, and emphasis <Integer>.

fontDirectoryIncludesFontFileNamed: aString

Return true if the file named aString is in the font directory.

fontFileStream: aString
Return a file stream for the font named aString.

fontNames: anArray
Answer an Array of StrikeFonts corresponding to anArray of String names. Load or synthesize the fonts, if necessary, and register the elements of the array in the FontManager.

install: aString
Install the font named aString if necessary. Load or synthesize the font if necessary. Complain if the font is missing.

install: aString **ifAbsent:** aBlock
Install the font named aString if necessary. Load or synthesize the font if necessary. Answer the result of evaluating aBlock if the font is missing.

virtualFontNames: anArray
Answer an Array of StrikeFonts or VirtualStrikeFonts corresponding to anArray of font names, and register the elements of the array in the FontManager.

virtuallyInstall: aFontName
Virtually install the font named aFontName if necessary. Create virtual fonts to load or to synthesize the font if necessary. Complain if the font cannot be constructed.

Class Methods

class initialization

initialize
Install the global FontManager, if none exists. Create an Emphases dictionary of associations between the emphasis string in a font name and the corresponding emphasis code.

Rationale

StrikeFontManager insures that all the **StrikeFonts** you reference are recorded in one place. Once a **TextStyle** is installed and used, its **StrikeFonts** are recorded in the **StrikeFontManager**. The **StrikeFontManager** keeps track of whether a **StrikeFont** has been loaded in the image or whether it is a **VirtualStrikeFont** and needs to be loaded when it is used the first time. When a **StrikeFont** is registered with the manager, it is accessible throughout the system. There are 64 fonts of compatibility types 1, 2, and 3 plus several other fonts in the directory returned by OS fontDirectory. When you bring up the standard image the first time by typing `smalltalk`, the **FontManager** has the **StrikeFonts** needed by the default text style.

Discussion

The global **FontManager** is used in Smalltalk code to refer to the one instance of **StrikeFontManager** that will exist at all times. It is possible to create a new instance of this class, but you are not likely to do so.

Instance Protocol

Accessing

Various methods are provided to access a **StrikeFont** in the dictionary, put a **StrikeFont** into the dictionary, and return an array of **StrikeFonts** after loading or synthesizing them (if necessary). You can register a **StrikeFont** without actually loading it into the image by sending the message **virtuallyInstall:**.

You are more likely to use a **TextStyleManager** instance creation message to load a **StrikeFont** than to use one of these accessing messages. It is reasonable, however, to add a font to the dictionary under these cases:

- You have modified a font, for example, by changing characters.
- You have modified a font by changing its emphasis (extra bold underlined, for example).
- You rename the font.

You might have occasion to use the messages **virtualFontNames:**, **fontNames:**, and **install:**. **FontNames:** guarantees that the fonts are installed in the image — this makes your image bigger. **Install:** is easier to use than the other two since it takes a string, not an array, as an argument.

Related Classes

StrikeFont
TextStyle
TextStyleManager
VirtualStrikeFont

Object subclass: #StructOutputTable

instanceVariableNames:	'globalDict mapArray idCount '
classVariableNames:	''
poolDictionaries:	''
category:	'System-Support'

Summary

StructOutputTable is a table used to store mappings for an object. This mapping detects and preserves the cycles of circular objects. This table is used in the process of storing objects in an external format (called a structure), usually invoked by the message **storeStructureOn:** or **storeStructureOnFile:**.

Instance Variables

globalDict <Dictionary>
A dictionary containing all unique values in the Smalltalk dictionary. Because these values are unique for each Smalltalk image, they are not written out.

idCount <Integer>
This instance variable is used to assign identification numbers to objects as they are written out. It is incremented as each new object gets an identification number assigned.

mapArray <Array>
An array of subcollections pairing the identification number derived from idCount with the object to which it is assigned. This pairing allows StructOutputTable to reference previously written objects instead of rewriting them, thus allowing the methods that write structures to deal with circularity.

Instance Methods

initialize-release

new: arraySize **globalDict:** dict
Initialize the receiver.

accessing

idOfElement: anObject **ifNew:** aBlock
Answer the integer ID of anObject in the receiver's structure, evaluating aBlock if the object has not been previously encountered.

if: anObject **isGlobal:** aBlock
Evaluate aBlock if anObject is in the receiver's global dictionary.

Class Methods

instance creation

new
Answer a new instance of the receiver.

Rationale

The Smalltalk compiler has a limit on the size of objects it can reconstruct. Also, methods which rely on the compiler cannot recognize the circularity of objects. **StructOutputTable** provides a means for objects exceeding the size limit, or circular objects, to be stored externally and transferred between images.

Background

The messages **fileOut:** and **fileIn:** can only be used with objects in a specific format. This format is one the compiler can read.

Other methods which use the compiler format, such as **storeOn:**, cannot correctly file circular objects into or out of an image, because they do not keep track of which objects they have already handled. Objects can contain references (direct or indirect) to themselves. This circularity can send these methods into an infinite loop.

Unlike compiler-based methods, the messages **storeStructureOn:** and **readStructureFrom:** make use of **StructOutputTable** to keep track of each structure as it is processed. Using this class, it is possible to move large circular objects into and out of an image.

Discussion

Discussion of the methods for copying Smalltalk structures may be found in Section 5, Programming in Smalltalk, of the manual *Tektronix Smalltalk Users.*

The class protocol *instance creation* includes one method which returns a new instance of the receiver, ready for use by the structure-storing methods.

The instance protocol *initialize-release* includes one method which creates a new array and reads the Smalltalk global variables into the instance variable **globalDict.** It is called by the instance creation method.

The instance protocol *accessing* includes the method **idOfElement:ifNew:** which evaluates whether an object has been encountered previously. If it has, it returns the identification number of the object. If it has not, it creates a unique identification number for the object passed in and stores the object-ID pair in the **mapArray**. This is how **StructOutputTable** keeps track of circularities. *Accessing* also includes the method **If:isGlobal:** which determines if the object is in the global dictionary. If it is, the method evaluates the block passed in. These methods are called every time an object that is part of the structure is written.

Examples

Forms can be written in a format readable by the compiler, using the message **storeOn:**, however, they can also be written in structure format. The following example code uses the method **storeStructureOn:** to write a **Form** out to the disk, and the method **readStructureFrom:** to read the **Form** back into the image. It can be executed in a workspace. In doing so, the class **StructOutputTable** is used.

```
aFileStream ← Disk file: 'example.struct'.
Form fromUser storeStructureOn: aFileStream.   "write it out"
aFileStream reset.   "move the file pointer back to the start"
newForm ← Object readStructureFrom: aFileStream.   "read in"
aFileStream close.   "clean up"
newForm display   "prove that it worked"
```

Related Classes

Methods which write structures using **StructOutputTable** are implemented in class **Object**. **Object** is also the class which contains methods to read the structures. Classes with unusual format, such as **ContextPart**, override certain internal methods.

StructureArray

```
ExternalPointerData subclass: #StructureArray

    instanceVariableNames:      'numberOfElements elementClass '
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

StructureArray represents an array of **ExternalData** structures which may contain embedded pointers. Each element must be an instance of the same structure. **StructureArray** provides protocol for accessing and updating elements and protocol for creating instances.

Subclasses can be made to provide protocol for accessing, by name, fields from an element of an instance.

Instance Variables

elementClass <Class>
The class of each element of the array.

numberOfElements <Integer>
The number of elements in the array.

Instance Methods

accessing

at: anIndex
Return the instance of elementClass at anIndex.

at: anIndex **put:** anElem
Update the receiver's element at anIndex with anElem.

elementClass
Return the class of elements of the receiver.

elementNumberOfPointers
Answer the number of pointers in each element.

elementPointersSize
Answer the pointersSize of each element.

elementSize
Answer the size of each element.

numberOfElements
Answer the number of elements in the receiver.

size
Return the size of the receiver.

Class Methods

instance creation

new: numberOfElems **class:** aClass
Return an instance capable of holding numberOfElems elements of class, aClass.

Rationale

StructureArray is used by the system call class for your operating system to pass arguments to system calls which use arrays of structures. Examples of such system calls include *readv(2)* and *writev(2)*.

Related Classes

UTekSystemCall uses this class.

Subtask


```
Object subclass: #Subtask
    instanceVariableNames:      'accessProtect arguments environment
                                exceptionValue initBlock priority program status
                                taskID terminateBlock terminationValue
                                waitSemaphore '
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Interface'
```

Summary

An instance of **Subtask** represents a spawned operating system task. Its parent task is the Smalltalk task. For clarity, threads of control in the operating system are referred to as "tasks", and threads of control in Smalltalk are referred to as "processes". **Subtask** contains protocol for testing and controlling the spawned task. A subtask can be waited for in a manner which suspends the entire Smalltalk parent task, or in a manner which suspends only the controlling Smalltalk process. The **Subtask** metaclass contains protocol for the management of spawned subtasks.

Instance Variables

accessProtect <Semaphore>

A semaphore used to protect accessing and setting of the status instance variable.

arguments <OrderedCollection>

A collection of strings, each of which is an argument to the program.

environment <Dictionary>

A dictionary of operating system environment variables, keyed by environment variable. Dictionary values are the values of the environment variables.

exceptionValue <Integer>

This instance variable contains the interrupt number of the signal causing termination. Non-zero values represent error conditions.

initBlock <Block>

A block to be executed between the fork call and the exec call. Usually this block involves signals and communication.

priority <Integer>
An integer representing the operating system priority of the subtask.

program <String>
A string containing the path of the program to be executed.

status <Symbol>
A symbol indicating the status of the task. Possible values:
nil
#running
#terminatedNormally
#waitedOn
#terminationSignaled
#terminatedWithCode
#terminatedWithSignal
#nonexistent.

Transition State Changes

Old State	Cause of Change	New State
nil	start	#running
#running	wait	#waitedOn
#running #waitedOn	successful interrupts	#terminationSignaled
#waitedOn	exit status > 0 signal status > 0 exit and signal status =0	#terminatedWithCode #terminatedWithSignal #terminatedNormally
any	snapshot	#nonexistent

taskID <Integer>
A unique identifying number assigned by the operating system.

terminateBlock <BlockContext>
A block containing code to terminate the spawned subtask. A value of nil means terminate using the default action, sending a terminate interrupt.

terminationValue <Integer>
This instance variable represents the code assigned by the exit system call. Non-zero values represent error conditions.

waitSemaphore <Semaphore>
A semaphore used to block the initiating process until the child task has terminated.

Instance Methods

initialize-release

initialize
Set the subtask data to reflect the parent's environment.

release
Remove the receiver from the list of scheduled subtasks.

accessing

arguments
Answer an OrderedCollection of arguments used in invoking this subtask.

enhancedPriority
Set the priority of the subtask to the highest possible priority.

environment
Answer the environment for this subtask in a dictionary format.

environment: envDictionary
Store the environment variables for this subtask. See 'environment variables' protocol in the metaclass.

priority
Answer the absolute priority of the subtask.

priority: aPriority
Set the priority of the subtask.

program
Answer the name of the executable program.

status
Answer the status of the receiver. Access with the protocol critical: to prevent inconsistencies in the status instance variable.

taskID
Answer an identifying unique integer assigned by the operating sytem.

terminateBlock
Answer the block that specifies the receiver's termination.

terminateBlock: aBlock
Record a block that allows the receiver to terminate in its own manner.

controlling

interrupt: anInterruptID
Send an interrupt to my task. AnInterruptID is a system dependent integer indicating which interrupt to send. The usual result of an interrupt is task termination.

start
Start the receiver by spawning a child, executing code to set up the child task (mainly communication and signal processing), and executing the program. If the execute fails terminate the child task. The child task will inherit the priority of the Smalltalk process.

terminate
Attempt to terminate the receiver's task in a manner which can be intercepted.

terminateUnconditionally
Terminate this task unconditionally.

wait
Wait for the receiver to terminate.

waitWithSmalltalkSuspended
Bypassing the dead child signal management, a wait system call is made. This call suspends the execution of the Smalltalk task. The list of scheduled subtasks is updated accordingly and the status of the termination is recorded for each child process until the receiver's subtask is found. This method replaces the use of the wait message, and is designed for use with non-interactive tasks that require the shutting down of the Smalltalk process for efficiency.

copying

copy
Inappropriate for a Subtask.

testing

abnormalTermination
Answer true if the status indicates abnormal termination, otherwise false.

isActive
Answer a boolean indicating if the receiver is running.

isNonExistent
Answer a boolean indicating if the receiver's task is not present.

isTerminated
Answer a boolean indicating if the receiver has been terminated.

notTerminated
Answer a boolean indicating if the receiver has not been terminated.

```
Subtask class

   instanceVariableNames:      'brokenPipesProcess scheduledSubtasks
                               scheduledSubtasksAccessProtect
                               unscheduledSubtasks waitProcess '
```

Subtask class — Instance Variables

brokenPipesProcess <Process>
This process runs forever, forking error notifiers upon signal receipt.

scheduledSubtasks <Dictionary>
Contains all the current subtasks keyed by taskID. Subtasks are added to the list when started. Management of these tasks is done by the metaclass.

scheduledSubtasksAccessProtect <Semaphore>
Semaphore for mutual exclusion to protect accessing of list of currently scheduledSubtasks.

unscheduledSubtasks <Dictionary>
Contains unscheduled subtask termination information keyed by taskID. This information, in the form of an executed wait system call, is collected and saved by the subtask management system. It is produced when a task dies, and it is not in the scheduled task list.

waitProcess <Process>
This process runs forever. Each time a "dead child" signal is received, a wait system call is made and the subtask management information updated.

Class Methods

class initialization

initialize
Create the accessing semaphore. Using the accessing semaphore, create a new dictionary of scheduled subtasks.

install
This method is invoked when resuming after a snapshot. When resuming, connections to the task controlling and signaling mechanisms in the operating system must be re-established. Create these connections by forking processes to catch the dead child signal and the broken pipe signal. Also, create and initialize a new list of scheduled subtasks.

installBrokenPipeProcess
Install a process to catch broken pipe signals. Each time the signal is received, fork an error notifier.

installSubtaskTerminationProcess
Install a process to monitor spawned subtasks. If a signal indicating a child task termination is received, update the current data about spawned subtasks. If possible (wait call is non-blocking), continue updating until there are no more terminated subtasks.

environment variables

currentEnvironment
Answer the current environment. For modification on a per subtask basis, senders should copy.

initializeEnvironment
Initialize the internal record of the environment with which this image was invoked.

instance creation

fork: commandName **then:** aBlock
Return a new instance of Subtask containing all the necessary information to execute the subtask. There are no arguments and the initialization block is empty.

fork: commandName **withArguments:** arguments
Return a new instance of Subtask containing all the necessary information to execute the subtask. Arguments is an OrderedCollection of arguments to the executable program specified by commandName. The initialization block is empty.

fork: commandName **withArguments:** arguments **standardIn:** in **standardOut:** out **standardError:** err
Return a new instance of Subtask containing all the necessary information to execute the subtask. Arguments is an OrderedCollection of arguments to the executable program specified by commandName. Three pipes are specified by the arguments in, out and err. Unused ends of the pipes are closed in the child process only. Senders must close the pipes for the

parent task.

fork: commandName **withArguments:** arguments **standardIn:** in **standardOutAndError:** out
Return a new instance of Subtask containing all the necessary information to execute the subtask. Arguments is an OrderedCollection of arguments to the executable program specified by commandName. Two pipes are specified by the arguments in and out. Unused ends of the pipes are closed in the child process only. Senders must close the pipes for the parent task.

fork: commandName **withArguments:** arguments **then:** aBlock
Return a new instance of Subtask containing all the necessary information to execute the subtask. Arguments is an OrderedCollection of arguments to the executable program specified by commandName. ABlock is an initialization block executed by the spawned task upon startup.

fork: commandName **withArguments:** arguments **withEnv:** anEnvironment **then:** aBlock
Return a new instance of Subtask containing all the necessary information to execute the subtask plus specification of a list of environment variables. Arguments is an OrderedCollection of arguments to the executable program specified by commandName. ABlock is an initialization block executed by the spawned task upon startup.

scheduled subtasks

addSubtask: aSubtask
Add a subtask to the dictionary of scheduled subtasks. Check the unscheduled subtask list to see if the task terminated before this method was called. If it terminated, update the task accordingly.

addUnscheduledSubtask: aSubtaskID **with:** syscall
Add a subtask to the list of unscheduled subtasks.

locateSubtask: aSubtaskID
Given a subtask's ID, answer the subtask object stored in the dictionary of subtasks.

removeSubtask: aSubtask
Remove a subtask from the dictionary of subtasks.

scheduledSubtasks
Answer a dictionary of subtasks, keyed by subtask ID.

task management

markAndSignalAll
Mark the status #nonexistent and signal the wait semaphore of each previously scheduled subtask. This method is used when restoring after a snapshot.

terminate: aTask
Terminate aTask and remove it from the task list.

terminateAll
Terminate all the scheduled tasks and remove them from the task list.

terminateUnconditionally: aTask
Terminate the spawned task and remove it from the task list.

Rationale

Tektronix Smalltalk adds support for subtasks to make the job of creating, running, and communicating with the subtasks straightforward. Interfaces to operating system signals, program parameters, environment variables, and subtask priorities are also supported.

Background

Multi-tasking

Most operating systems support multi-tasking. To the operating system, your running Smalltalk program is another task. Although Smalltalk has its own processes, Smalltalk can also create and communicate with operating system tasks.

Operating system subtasks provide access to other executable programs. By using the class **Subtask**, you can create a child task so that you can use some resource — an operating system command, or a program you have written in the C language, etc. — outside of the Smalltalk process.

By spawning new tasks, multi-tasking is accessible without leaving the Smalltalk environment. A newly spawned task is called a child task or a subtask. The original task is referred to as the parent task. The child task is a "copy" of the parent task — it shares resources with the parent task. Since only one task can execute at a time, CPU time is also shared, initially in a predetermined fashion. Common practice is for the spawned child task to perform some chore, and then report back to the parent task. After reporting, the child task terminates. A parent may choose to relinquish use of the CPU until a subtask terminates. It does this by an operation called waiting. While waiting, the parent task is blocked and cannot do anything else until the child task terminates.

The child task's chore is often accomplished by finding some other program to do the work. The use of this other program is known as an exec operation (for execute). In an exec operation, the spawned task "turns itself into" the other program.

Pipes

A pipe is used to convey data from one process (task) to another. Pipes are data structures set up in computer memory to be transient, even though they share other characteristics with files.

Frequently, a parent task may want to communicate with a child task. Information can be sent to and from the child task by using pipes; however, each pipe can send information in only one direction. If communication in two directions is desired, two pipes must be used. Pipes are similar to files with two critical differences.

- Files can be reopened many times. Pipes can only be opened once. Once a pipe is closed it is gone.
- Files can be reset and repositioned. It is not possible to reposition a pipe.

Usually the parent task creates a pipe. Each end of the pipe is assigned a file descriptor, one for reading and one for writing. When a subtask is created it inherits these open file descriptors. The parent task saves one file descriptor, the one which is appropriate for its direction of communication. For example, if the parent task wants to send information to the child, the parent saves the file descriptor for writing. Since the parent will not be using the reading end of the pipe, it should close this unused end. The child task must also save the appropriate file descriptor and close the file descriptor corresponding to the unused end of the pipe. Neglecting to close these unused pipe file descriptors might mean the task could run out of file descriptors, since there is a limit on the number of open file descriptors per task.

Sometimes it is not possible for the child task to know that it should use the pipe's file descriptors for reading and writing. For instance, the child task might execute a program that writes on standard output. It is possible for the child task to redirect its I/O by mapping its pipe descriptors to known file descriptors. Redirection allows a file descriptor to capture all the data intended for another descriptor. Tektronix Smalltalk **Pipe** protocol supports this functionality. Once a pipe's file descriptor is mapped, it becomes obsolete and should be closed. For example, the child task may want to write to the pipe, but the program is designed so write operations go to standard output. The write file descriptor of the pipe must be mapped to standard output's file descriptor (1), and the pipe's original write file descriptor should be closed. The effect of the mapping in this example is for the child task's write operations going to standard output to be performed on the write end of the pipe

instead.

Operating System Multi-tasking — Implementation

In a multi-tasking operating system, programs generally execute by duplicating the parent program (task), transforming the duplicate (child) task into the new program. Upon termination of the new program, the parent task is signaled; if it suspended execution, the parent task resumes. Specific system calls are used to accomplish these tasks. A *fork* call causes the duplication of the parent task. An *exec* call causes the duplicate task (i.e., the child task) to "transform" into a desired executable program. An *exit* call terminates the sending task and causes the operating system to send a "dead child signal" to the parent task — this indicates that a spawned task has terminated. A *wait* call executed by the parent task, besides suspending the parent task until termination of the child task, returns the termination status of the child task. Termination status includes information such as which signal caused the termination and whether the termination was abnormal.

For example, suppose you type the command *ls* at the keyboard. As you know, the shell program is waiting to interpret your keystrokes. Here the shell is a parent task. If the shell determines that you have typed *ls* correctly, it forks another shell task — a child task. This task then executes an *exec* call which overlays the child task with the *ls* program. *ls* executes, outputs directory information to the screen, and exits normally. The shell task receives the termination status via a *wait* call and resumes its I/O wait for more input from you.

Discussion

A subtask is spawned in a Unix-like operating system in three phases:

- a fork system call,
- a set-up phase, and
- the exec system call.

Most of the code for what takes place in the set-up phase is encapsulated in the subtask at the point of instance creation. Between fork and exec the subtask is running Smalltalk, but it is limited. The keyboard and the mouse cannot be used to communicate with Smalltalk, so no debugging can take place if the set-up code doesn't execute as anticipated.

In the set-up phase, depending upon which instance creation message selector is used and the values of instance variables, several things can take place, such as:

- a block of code executes concerning signals and communication;
- priority of the subtask is set;
- the environment is changed from the environment that the subtask inherited from its parent task.

Subtask Instance Protocol

Accessing methods enable the user to set and access the values of instance variables. Some instance variables control what happens in the set-up phase, others contain information about the termination of the subtask.

Controlling methods start the subtask, wait for the subtask, and allow the task to be interrupted or terminated.

Testing methods provide information about the subtask, such as whether it is active, terminated abnormally, is or is not terminated, and whether the subtask exists.

Subtask Class Protocol

Environment variables methods provide information about the current environment and allow the environment of the subtask to be modified. The default environment for a subtask is the environment of the parent task.

Instance creation methods provide the encapsulation of data to be used in the set-up phase of the subtask.

Scheduled subtasks and *task management* methods deal with the subtask management system and you will probably not use them directly.

Relation to the System Call and Pipe Classes

The system call class is used for the implementation of fork, exec, and wait. It also contains protocol for various other operations used by a subtask, including setting up signals and priorities. Communication between tasks is currently provided by the pipe classes. Access to other communication implementations, such as sockets, is available through system calls.

Management of Subtasks

A subtask management process is part of Tektronix Smalltalk — it is implemented in the metaclass of **Subtask**. Unlike the C programming environment where the only type of waiting available is complete suspension until the child task terminates, the task management system allows two types of waiting, described below.

The subtask manager runs continuously, monitoring child tasks. When a task is started, it is registered with the task management system. Two lists are used to keep track of subtasks — class instance variables **scheduledSubtasks** and **unscheduledSubtasks.**

When a subtask terminates, the manager receives a dead child signal. The manager releases a terminated task from its list; if the parent process has been suspended, the manager sends a signal for it to resume.

Waiting

There are two methods that deal with waiting by the parent task — **wait** and **waitWithSmalltalkSuspended.** The message **wait** causes the suspension of only the parent process of the subtask. When the subtask ends, the parent is signaled to resume.

The message **waitWithSmalltalkSuspended** causes the entire Smalltalk task to suspend. While Smalltalk is suspended there is no way to interact with the Smalltalk task using the keyboard or the mouse. This version of waiting is used for efficiency, for example, while a Fortran program doing a lot of calculations is running — like a fast Fourier transform.

Concurrent subtasks may use either form of waiting — their terminations are handled appropriately by the subtask manager.

Snapshots

If a subtask is running when a snapshot is made, certain things occur. In the Smalltalk image that continues to run after the snapshot, the subtask is not affected.

When you quit, any running subtasks are terminated by the operating system. When a Smalltalk image is "brought up" (loaded by the Smalltalk interpreter), the subtask management system is installed with empty task lists. If subtasks were running when the snapshot was made, they will be marked as #nonexistent. Their status can be checked and they can be restarted by application programs.

Examples

See the file */usr/lib/smalltalk/fileIn/Examples-Subtasking.st* (this path correct for UTek only) for some examples illustrating how to use **Subtask** in an application example. Read further for an introduction to using **Subtask.**

The Simplest Example

Here is a very simple example that uses **Subtask.**

```
"Execute a simple binary program with no arguments."

    | task |
    task ← Subtask
        fork: '/usr/bin/pretend'
        then: [ ].
    task start.
    task wait
```

The code above can be executed in a workspace, assuming that the "/usr/bin/pretend" file exists. It contains the simplest form of subtask instance creation, since it has no arguments and does not include a block of code to be executed between fork and exec. The default set-up takes place, not a user-specified set-up. The task then starts and the parent waits for the subtask to terminate.

Add Some Interest

Here is a method which executes a program requiring two arguments. It illustrates some variations of **Subtask** protocol and adds error checking code.

```
executeUtility: aCommand withArgument1: argumentString1
    withArgument2: argumentString2

"Execute a binary program with two arguments. Set the priority
of the subtask to the highest possible, and ignore dead child
signals in the child task. Create an error if the program cannot
be executed or if the program terminates abnormally."

    | argumentList task envDictionary |
    argumentList ← OrderedCollection
                    with: argumentString1 with: argumentString2.
    task ← Subtask
        fork: aCommand
        withArguments: argumentList
        then: [OS ignoreInterrupt: OS deadChildInterrupt].
    envDictionary ← Subtask currentEnvironment copy.
    envDictionary at: #PARENT put: 'smalltalk'.
    task environment: envDictionary.
```

```
task enhancedPriority.
task start isNil
   ifTrue: [self error: 'Cannot execute ', aCommand].
Cursor execute
   showWhile: [task wait].
task abnormalTermination
   ifTrue: [self error: 'Abnormal termination from ', aCommand]
```

In the code above, we begin by placing the arguments in an OrderedCollection, since the instance creation method expects arguments in that form. This example uses an instance creation message different from the first example — this one allows us to pass arguments to the executable program.

Set-up Phase

The block of code after **then:** is executed only by the child task; it is done in the set-up phase. Here the set-up changes the action of an interrupt — the dead child interrupt is ignored. The current environment is stored in a temporary variable, *envDictionary*, and the association of **#Parent->'smalltalk'** is added to it. Then the subtask's **environment** instance variable is set to the value of the temporary variable, *envDictionary*. The subtask is given the highest possible priority. The **start** message causes the execution of the set-up code.

Start and Finish

If **start** returns an error, a notifier is displayed. While the subtask is executing and the parent process waits for the subtask to terminate, the cursor is in the form of a star (Cursor execute showWhile:). If the subtask terminates abnormally, an error notifier is displayed.

Related Classes

In Smalltalk code, the typical reference to the system call class is **OS**, a global variable which is the appropriate system call class for your operating system. You might want to refer to these classes in this manual for further information:

- the system call class for your operating system and its superclasses,
- the **PipeStream** hierarchy of classes, and
- **Pipe.**

Pipe is not directly used in the implementation of **Subtask**, but it is essential to complete usage of subtasks.

```
ExternalBinaryData variableByteSubclass: #Tchars

    instanceVariableNames:      ''
    classVariableNames:         'BrkcDataIndex EofcDataIndex IntrcDataIndex
                                QuitcDataIndex StartcDataIndex StopcDataIndex
                                '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

Tchars provides accessing protocol for the following C structure.

```
struct tchars {
    char    t_intrc;    /* interrupt */
    char    t_quitc;    /* quit */
    char    t_startc;   /* start output */
    char    t_stopc;    /* stop output */
    char    t_eofc;     /* end-of-file */
    char    t_brkc;     /* input delimiter (like nl) */
}
```

The structure is documented under *tty(4)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

BrkcDataIndex

EofcDataIndex

IntrcDataIndex

QuitcDataIndex

StartcDataIndex

StopcDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **BrkcDataIndex** holds the offset of the "t_brkc" field.

Instance Methods

accessing

brkc
Return the value of the structure field named brkc.

brkc: aCharacter
Assign the argument, aCharacter, to the structure field named brkc.

eofc
Return the value of the structure field named eofc.

eofc: aCharacter
Assign the argument, aCharacter, to the structure field named eofc.

intrc
Return the value of the structure field named intrc.

intrc: aCharacter
Assign the argument, aCharacter, to the structure field named intrc.

intrc: iCharacter **quitc:** qCharacter **startc:** startCharacter
stopc: stopCharacter **eofc:** eCharacter **brkc:** bCharacter
Assign values to all the fields of the structure.

quitc
Return the value of the structure field named quitc.

quitc: aCharacter
Assign the argument, aCharacter, to the structure field named quitc.

startc
Return the value of the structure field named startc.

startc: aCharacter
Assign the argument, aCharacter, to the structure field named startc.

stopc
Return the value of the structure field named stopc.

stopc: aCharacter
Assign the argument, aCharacter, to the structure field named stopc.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

default
Return an instance containing the default characters.

Rationale

The structure is used in support of the following UTek system call:

ioctl(2)

Related Classes

UTekSystemCall implements the system call listed above.

```
Object subclass: #TextStyle
    instanceVariableNames:      'fontArray lineGrid baseline alignment firstIndent
                                restIndent rightIndent tabsArray marginTabsArray
                                outputMedium lineGridForLists baselineForLists
                                lineGridForMenus baselineForMenus '
    classVariableNames:         ''
    poolDictionaries:           'TextConstants '
    category:                   'Graphics-Support'
```

Summary

An instance of **TextStyle** is a grouping of fonts that "look nice together" and display characteristics used in composing text in these fonts.

Instance Variables

alignment <Integer>

Indicates the mode for placement from the margins:
0 = flush left, 1 = flush right, 2 = centered, 3 = justified.

baseline <Integer>

The amount to be added to the top of a line to find the baseline of the line. The baseline is the point from which the ascent of a font should rise.

baselineForLists <Integer>

Copied into baseline while constructing a TextStyle for a list.

baselineForMenus <Integer>

Copied into baseline while constructing a TextStyle for a menu.

firstIndent <Integer>

Amount to inset from the left margin for the first line of a paragraph. Initial value for paragraph associated with this TextStyle.

fontArray <Array>

A collection of fonts available in this TextStyle. These may be either StrikeFonts or VirtualStrikeFonts. The emphasis portion of a Text (the runs instance variable) returns a value for indexing into the fontArray.

lineGrid <Integer>

The amount to be added to the top of a line to find the top of the next line.

lineGridForLists <Integer>
Copied into lineGrid while constructing a TextStyle for a list.

lineGridForMenus <Integer>
Copied into lineGrid while constructing a TextStyle for a menu.

marginTabsArray <Array>
Each value in the array is a tuple indicating inset values to tab to relative to the left and right margin of this paragraph. Allows for inset paragraphs.

outputMedium <Symbol>
Currently only #Display is supported.

restIndent <Integer>
Amount to inset from the left margin for all but the first line of a paragraph. Initial value for paragraph associated with this TextStyle.

rightIndent <Integer>
Amount to inset from the right margin for all the lines of the paragraph. Initial value for paragraph associated with this TextStyle.

tabsArray <Array>
Tab stops. Values are relative to the left margin of the paragraph.

Pool Dictionaries

TextConstants
A dictionary of symbols for non-printing characters, symbols related to text composition and text emphasis, and default values for text composition and text emphasis.

Instance Methods

accessing

alignment
Answer the code for the current setting of the alignment.

alignment: anInteger
Set the current setting of the alignment to anInteger — 0=flush left, 1=flush right, 2=centered, 3=justified.

baseline
Answer the distance from the top of the line to the bottom of most of the characters (by convention, bottom of A).

baseline: anInteger
Set the distance from the top of the line to the bottom of most of the characters.

baselineForLists
Answer the baseline for a list composed from this TextStyle.

baselineForLists: anInteger
Set the instance variable baselineForLists.

baselineForMenus
Answer the baseline for a menu composed from this TextStyle.

baselineForMenus: anInteger
Set the instance variable baselineForMenus.

defaultFont
Answer the first font in fontArray.

descent
Answer the distance from the bottom of most of the characters (by convention, bottom of A) to the top of the next line.

firstIndent
Answer the horizontal indent of the first line of a paragraph in the style of the receiver.

firstIndent: anInteger
Set the horizontal indent of the first line of a paragraph in the style of the receiver.

fontArray
Answer the fontArray of this TextStyle.

fontArray: aFontArray
Install aFontArray of fonts; recompute lineGrids and baselines.

fontAt: index
Return the StrikeFont at index. Make sure whenever a font is accessed, it is coerced into a StrikeFont and registered in the global FontManager. Recompute the listStyle and menuStyle based on the font returned.

fontAt: index **put:** font
Set the fontArray element at index to font.

fontFor: fontIndex **emphasis:** emphasisBlock
Select and return the index of the first font in fontArray which has the same size and family as the font at fontIndex, and where emphasisBlock evaluates true.

fontFor: fontIndex **face:** face
Select and return the index of the first font in fontArray which has the same size and face as the font at fontIndex.

fontNamed: aString
Return the font named aString. If it is not found in fontArray, return the basal font for the TextStyle.

lineGrid
Answer the relative space between lines of a paragraph in the style of the receiver.

lineGrid: anInteger
Set the relative space between lines of a paragraph in the style of the receiver.

lineGridForLists
Answer the relative space between lines of a list in the style of the receiver.

lineGridForLists: anInteger
Set the relative space between lines of a list in the style of the receiver.

lineGridForMenus
Answer the relative space between lines of a menu in the style of the receiver.

lineGridForMenus: anInteger
Set the relative space between lines of a menu in the style of the receiver.

nestingDepth
Return the number of entries in the marginTabsArray.

outputMedium
Answer the outputMedium for this style.

outputMedium: aSymbol
Set the outputMedium for this style (currently only #Display is recognized).

restIndent
Answer the indent for all but the first line of a paragraph in the style of the receiver.

restIndent: anInteger
Set the indent for all but the first line of a paragraph in the style of the receiver.

rightIndent
Answer the right margin indent for the lines of a paragraph in the style of the receiver.

rightIndent: anInteger
Set the right margin indent for the lines of a paragraph in the style of the receiver.

upperLead: upperLeadInteger **lowerLead:** lowerLeadInteger
Collect all fonts that are in the largest point size in this TextStyle. Use this subset of the fonts to compute the appropriate baseline and line gridding including the additional leading amounts specified.

converting

asListStyle
Answer a copy of the receiver with lineGrid and baseline set for lists.

asMenuStyle
Answer a copy of the receiver with lineGrid and baseline set for menus.

tabs and margins

clearIndents
Reset all the margin (index) settings to 0.

leftMarginTabAt: marginIndex
Set the 'nesting' level of left margin indents of the paragraph in the style of the receiver to marginIndex.

nextTabXFrom: anX **leftMargin:** leftMargin **rightMargin:** rightMargin
Tab stops are distances from the leftMargin. Answer either the first tab stop after anX (normalized relative to leftMargin) or rightMargin, whichever is greater.

rightMarginTabAt: marginIndex
Set the 'nesting' level of right margin indents of the paragraph in the style of the receiver to marginIndex.

tabWidth
Answer the width of standard tab.

Class Methods

constants

default
Answer the system default text style.

default: aTextStyle
Change the system default text style to aTextStyle.

examples

allDefaultFontNames
When you see the star and arrow cursor, select a point (by clicking a mouse button) for each font in the DefaultTextStyle. Each font in the default text style will be displayed.

defaultStyleHi
When you see the star and arrow cursor, select a point by pressing a mouse button. The greeting will be displayed there in the system default text style.

instance creation

fontArray: anArray
Return a TextStyle constructed from the fonts given in anArray.

Rationale

A **TextStyle** is used for text composition. It includes a grouping of fonts that "go together", often from the same family, and look attractive together interspersed in text. In addition to one or more fonts contained in the instance variable **fontArray**, a **TextStyle** contains information (**lineGrid**, **baseline**, and more) computed from the largest font in the array. The information aids in the physical layout of multiple lines of text. Additionally, you can specify things such as indentation, tabs, and alignment.

Discussion

If you are not familiar with some of the terminology in this class, refer to **StrikeFont** in this manual for an explanation of some of the terms.

Class Protocol

Ordinarily, you would not use the *instance creation* message, **fontArray:**, to create an instance of **TextStyle**. Instead, use a **TextStyleManager** *text style instance creation* message or send the message **asTextStyle** to a **StrikeFont**. If you want to create a text style for lists or menus in your application, you could use the **fontArray:** message, or use a **TextStyleManager** message if you want it to be generally available. The protocol in the **TextStyleManager** registers the text style when it is created, so that it is accessible to others. Read about the advantages of using the **TextStyleManager** under that class in this manual.

Constants methods allow you to set the default text style or get a copy of the current default text style.

Instance Protocol

Accessing methods allow you to set or access the values of the instance variables. Additionally, methods install a font or array of fonts, set an element in **fontArray** to the supplied font, or return the index of a font matching a supplied fontIndex' size, family, face, or emphasis. One method adjusts the **lineGrid** and **baseLine** to include supplied "leading" at the top and bottom of the characters.

Converting methods return a copy of the **TextStyle** with **lineGrid** and **baseLine** adjusted for lists or menus.

Tabs and margins methods clear all the indents, set the left and right "nesting" level of indents, and answer the **DefaultTabWidth** (from the **TextConstants** pool dictionary). The method **nextTabXFrom:leftMargin:rightMargin:** answers either the first tab stop after a place on the line specified by the user or the right margin, whichever is greater.

Examples

The following method is in the **TextStyle class** *examples* message category.

```
defaultStyleHi

"When you see the star and arrow cursor, select a point by pressing a mouse button.
The greeting will be displayed there in the system default text style."

   (DisplayText text: 'Hi there!\How are you?' withCRs asText
              textStyle: (TextStyle default))
   displayAt: Sensor waitClickButton
```

First, the *instance creation* message **text:textStyle:** is sent to **DisplayText**. The first argument must be a **Text**, so the string is converted to a **Text**. The message **withCRs** was sent to the string to replace the backslash (\) with a carriage return in the string. The second argument is a **TextStyle** — the one returned by the **default** message is the value at DefaultTextStyle in the **TextConstants** dictionary. The **DisplayObject** message **displayAt:** is sent to the instance of **DisplayText** with the argument of the **Point** created by Sensor waitClickButton. **Sensor** is a global variable, an instance of **InputSensor.**

The following method is in the **TextStyle class** *examples* message category.

```
allDefaultFontNames

"When you see the star and arrow cursor, select a point (by clicking a mouse button) for
each font in the DefaultTextStyle. Each font in the default text style will be displayed."

   (TextConstants at: #DefaultTextStyle) fontArray do:
   [: strikeFont |
   (DisplayText text: strikeFont name asText textStyle: strikeFont asTextStyle)
   displayAt: Sensor waitClickButton]
```

This method takes **defaultStyleHi** one step farther, and makes use of additional **TextStyle** methods. TextConstants at: #DefaultTextStyle returns the same result as TextStyle default, used in the preceding example. The instance of **TextStyle** is sent the message **fontArray**, and the returned array of **StrikeFonts** is enumerated with a block similar to the code in the first example. Instead of the greeting, the argument to **text:** is the string returned by sending the message **name** to the **StrikeFont**. The argument to **textStyle:** is the **TextStyle** returned when the message **asTextStyle** is sent to the **StrikeFont**. When this example is executed in the standard image (assuming that you have not changed the default text style), 16 font names are displayed using a separate **TextStyle** for each one. One point of this example is that text must be converted to a displayable object, in this case a **DisplayText**, in

order to display it. In fact, all text that you see in Smalltalk is converted to a displayable object, but that is handled for you and you aren't required to do the converting in everyday use. Unless you want to deal with the **Forms** in the **glyphs** instance variable of a **StrikeFont**, the only way to see a **StrikeFont** is to make it a **TextStyle** and display some text using the **TextStyle**.

Related Classes

DisplayText
Paragraph
StrikeFont
StrikeFontManager
String
Text
TextStyleManager
VirtualStrikeFont

```
Dictionary variableSubclass: #TextStyleManager

    instanceVariableNames:      ''
    classVariableNames:         'MenuDependents TextStyleMenu
                                TextStyleNames '
    poolDictionaries:           ''
    category:                   'Graphics-Support'
```

Summary

This class is the central repository of all **TextStyles**. **TextStyleManager** maps **TextStyle** names (**Strings**) to **TextStyles**.

Class Variables

MenuDependents <OrderedCollection>

The list of MenuDependents is kept so that whenever the default text style changes, all cached menus are flushed. To add a menu to the list, send the message addMenuDependents: to the global StyleManager. Each element of MenuDependents is itself a collection of three elements:

- The symbol name of a Smalltalk class or other entry in the system dictionary.
- The symbol #class or #instance.
- The symbol name of a unary message selector.

TextStyleMenu <PopUpMenu>

A menu of available TextStyle selections.

TextStyleNames <Dictionary>

A dictionary of TextStyle names and associated TextStyles.

Instance Methods

accessing

at: aString **put:** aTextStyle

Install aTextStyle named aString.

removeAssociation: anAssociation **ifAbsent:** aBlock
Remove the key and value association, anAssociation, from the receiver. Cause all MenuDependents to be flushed. Answer anAssociation.

removeKey: aString **ifAbsent:** aBlock
Remove the TextStyle named aString else answer aBlock value.

default text style

changeDefaultTextStyle
Present a menu of TextStyles. If one is selected, change the default TextStyle.

changeDefaultTextStyle: aTextStyle
Change the default TextStyle to aTextStyle.

menu initialization

initializeMenus
Initialize all of the system menus. New classes cacheing menus or other dependencies upon TextStyle should be added to the list of MenuDependents by sending the message addMenuDependents: to TextStyleManager.

selecting

fromUser
Present a menu of TextStyles. Answer with the selected TextStyle. If no TextStyle is available or selected, return nil.

fromUser: aBlock
Present a menu of TextStyles. If one is selected, evaluate aBlock with the selected TextStyle and return the selected TextStyle. Return nil if no TextStyle is selected.

text style instance creation

styleName: aString **baseNames:** anArray
Create and install a TextStyle named aString with fonts specified by anArray of base String names. Load or synthesize the fonts if necessary. Set the leading to zero. Answer the TextStyle.

Each base name in anArray represents a set of eight fonts (Basal, Bold, Italic, BoldItalic, Basal Underlined, Bold Underlined, Italic Underlined, and BoldItalic Underlined). The font order within the new TextStyle is best explained by the following example:

```
StyleManager styleName: 'Pellucida Sans-Serif 10/12' baseNames:
#('PellucidaSans-Serif10' 'PellucidaSans-Serif12')
```

The code above generates a TextStyle with font order:

'PellucidaSans-Serif10'	(Basal)
'PellucidaSans-Serif10B'	(Bold)
'PellucidaSans-Serif10I'	(Italic)
'PellucidaSans-Serif10X'	(Bold Italic)
'PellucidaSans-Serif12'	(Basal)
'PellucidaSans-Serif12B'	(Bold)
'PellucidaSans-Serif12I'	(Italic)
'PellucidaSans-Serif12X'	(Bold Italic)
'PellucidaSans-Serif10U'	(Underlined)
'PellucidaSans-Serif10BU'	(Bold Underlined)
'PellucidaSans-Serif10IU'	(Italic Underlined)
'PellucidaSans-Serif10XU'	(Bold Italic Underlined)
'PellucidaSans-Serif12U'	(Underlined)
'PellucidaSans-Serif12BU'	(Bold Underlined)
'PellucidaSans-Serif12IU'	(Italic Underlined)
'PellucidaSans-Serif12XU'	(Bold Italic Underlined)

styleName: aString **baseNames:** anArray **lead:** leadInteger
Similar to styleName:baseNames:, but divide the leading equally between upper and lower leading (odd pixel on top).

styleName: aString **baseNames:** anArray **upperLead:** upperLeadInteger **lowerLead:** lowerLeadInteger
Similar to styleName:baseNames:, but set the upper and lower leading.

styleName: aString **fontNames:** anArray
Create and install a TextStyle named aString with fonts specified by anArray of explicit String names. Set the leading to zero. Answer the TextStyle.

styleName: aString **fontNames:** anArray **lead:** leadInteger
Similar to styleName:fontNames:, but divide the leading equally between upper and lower leading (odd pixel on top).

styleName: aString **fontNames:** anArray **upperLead:** upperLeadInteger **lowerLead:** lowerLeadInteger
Similar to styleName:fontNames:, but set the upper and lower leading.

Class Methods

class initialization

flushMenus
Set the TextStyleMenu to nil.

initialize
Install a new style manager if necessary. Build a new MenuDependents list.

initializeStyleManager
Install standard text styles into the StyleManager.

examples

bigHi
When you see the star and arrow cursor, move the cursor where you want it and press a mouse button. The greeting will be displayed in large Italic letters.

instance creation

new: anInteger
Flush the style menu because the new instance will probably be installed as StyleManager.

menu initialization

addMenuDependents: aCollection
Extend the list of MenuDependents by the list contained within aCollection. Each element of aCollection should itself be a collection of three elements:

- The symbol name of a Smalltalk class or other entry in the system dictionary.
- The symbol #class or #instance.
- The symbol name of a unary message selector.

When an instance of the receiver (typically the Smalltalk global StyleManager) receives the message initializeMenus, for each menu dependent the unary message is sent either to the class (#class) or to all instances of the class (#instance).

menuDependents
Answer the list of MenuDependents.

Rationale

This class maps names to **TextStyles** and provides a user-friendly interface to **TextStyles**, enabling the user to change the default text style, select from a menu of **TextStyles**, and add **TextStyles** to the menu. It keeps a list of **TextStyles** — once it is registered with the **TextStyleManager**, a **TextStyle** can be used anywhere in Smalltalk. When a **TextStyle** is removed from the menu, it will still be available to anything in Smalltalk that uses that **TextStyle**.

The **TextStyleManager** insures that when the default text style is changed all menu dependents are updated to the new default.

Discussion

Class Protocol

Ordinarily there is only one **TextStyleManager**, referred to by the global variable **StyleManager**. You are not prevented from creating an instance of **TextStyleManager** which is not the global manager. This implementation assumes, however, that a new **TextStyleManager** will be installed as **StyleManager**, so when the message **new:** is sent to the class, the **TextStyleMenu** is set to nil. Besides setting the **TextStyleMenu** to nil, *class initialization* includes a method to **initialize** this class.

Menu initialization methods allow you to add menu dependents or access **MenuDependents**. The message **addMenuDependents:** should be sent to the global **StyleManager** whenever you add a menu to the system, so that if the default text style is changed the menu(s) you added will be updated.

Instance Protocol

Accessing methods enable you to add a **TextStyle** to the manager, or remove a style from the manager by specifying either the String name of the **TextStyle** or the Association of a String name and a **TextStyle**. Either adding or removing a **TextStyle** causes the **TextStyleMenu** to be set to nil.

Default text style methods enable you to set the default text style by selecting one from the menu or specifying one.

Menu initialization contains one method to initialize all of the menus listed in **MenuDependents**. This message is sent by another method and is one you probably won't use.

Selecting contains two methods, **fromUser** and **fromUser:**, which return a **TextStyle** you select from a menu and evaluate a block with the selected **TextStyle**, respectively.

Text style instance creation includes a number of methods to create a **TextStyle** by providing a **String** name for the style (any name you prefer) and an **Array** of base **String** names (actual names of font families — found in the directory answered by OS fontDirectory fullName.). An alternative to base names is to provide an **Array** of fonts (either **StrikeFonts** or **VirtualStrikeFonts**). You also have the option of specifying "leading" to be divided equally at the top and bottom or separate "leading" at the top and bottom of a "line" of characters. Using the messages in this message category, **TextStyles** can be arbitrarily named, however, the convention is to use a name that reflects the grouping of fonts. The difference between the "baseName" and "fontName" methods is that the former load a group of eight fonts in a default order for each base name, the latter only load the fonts explicitly named in the array and in the order they are listed in the array.

The messages in *text style instance creation* are the preferred way to create text styles, instead of the **TextStyle class** instance creation message **fontArray:**. The messages here will register the new style with the manager.

Examples

The following method is in the **TextStyleManager class** *examples* message category.

```
bigHi

"When you see the star and arrow cursor, move the cursor where you want it and press
a mouse button. The greeting will be displayed in large Italic letters."
    | lines styleName |
    styleName ← 'BigStyle'.
    StyleManager styleName: styleName fontNames: #('PellucidaSans-Serif36I').
    lines ← 'Hi there!\How are you?' withCRs.
    (DisplayText text: (lines asText) textStyle: (StyleManager at: styleName))
        displayAt: Sensor waitClickButton.
    StyleManager removeKey: styleName ifAbsent: [ ].
```

First, a *text style instance creation* message is sent to the global **StyleManager** to create a **TextStyle**. On style menus its name is 'BigStyle'. 'BigStyle' has one font, Pellucida Sans-Serif 36I, in its **fontArray**. If a **TextStyle** is not registered with the manager, it cannot be referred to by name and cannot be accessed throughout Smalltalk.

Next, a **String** is created with two lines of characters — note the use of a backslash to indicate the carriage return and the message **withCRs**. The simplest way to display the string using a **TextStyle** is to make it a **DisplayText**. The instance creation message takes a **Text** and a **TextStyle** as arguments, then the **DisplayText** is displayed where you press a mouse button. Finally, 'BigStyle' is removed from the global **StyleManager**.

Related Classes

StrikeFont
StrikeFontManager
TextStyle
VirtualStrikeFont

```
ExternalBinaryData variableByteSubclass: #Timeval

    instanceVariableNames:      ''
    classVariableNames:         'SecDataIndex UsecDataIndex '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

Timeval provides creation and accessing protocol for the following C structure.

```
struct timeval {
    long    tv_sec;     /* seconds */
    long    tv_usec;    /* microseconds */
}
```

The structure is documented under *gettimeofday(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

SecDataIndex

UsecDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **SecDataIndex** holds the offset of the "tv_sec" field.

Instance Methods

accessing

sec
Return the value of the structure field named sec.

sec: anInt
Assign the argument, anInt, to the structure field named sec.

sec: anInt **usec:** anotherInt
Assign values to all the fields of the structure.

usec
Return the value of the structure field named usec.

usec: anInt
Assign the argument, anInt, to the structure field named usec.

converting

asTime
Return an instance of Time equivalent to the receiver.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

sec: anInt **usec:** anotherInt
Return an instance with the values of the fields assigned.

Rationale

The timeval C structure is used in support of these UTek system calls:

adjtime(2)
cfsettimeofday(2)
gettimeofday(2)
select(2)
settimeofday(2)
utimes(2)

Related Classes

UTekSystemCall implements the following system calls which use the timeval structure:

gettimeofday(2)
select(2)
utimes(2).

The other system calls which use this structure have not been implemented because they are only available to the superuser (root).

```
ExternalBinaryData variableByteSubclass: #Timezone

    instanceVariableNames:      ''
    classVariableNames:         'DsttimeDataIndex MinuteswestDataIndex '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

Timezone provides creation and accessing protocol for the following C structure.

```
struct timezone {
    int   tz_minuteswest;    /* minutes west of Greenwich */
    int   tz_dsttime;        /* type of dst correction */
}
```

The structure is documented under *gettimeofday(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

DsttimeDataIndex

MinuteswestDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **DsttimeDataIndex** holds the offset of the "tz_dsttime" field.

Instance Methods

accessing

dsttime

Return the value of the structure field named dsttime.

dsttime: anInt
Assign the argument, anInt , to the structure field named dsttime.

minuteswest
Return the value of the structure field named minuteswest.

minuteswest: anInt
Assign the argument, anInt , to the structure field named minuteswest.

minuteswest: anInt **dsttime:** anotherInt
Assign values to all the fields of the structure.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

minuteswest: anInt **dsttime:** anotherInt
Return an instance with the values of the fields assigned.

Rationale

The structure is used in support of the following UTek system calls:

cfsettimeofday(2)
gettimeofday(2)
settimeofday(2)

Related Classes

UTekSystemCall implements the following system call which uses the timezone structure:

gettimeofday(2).

The other system calls which use this structure have not been implemented because they are only available to the superuser (root).

```
AbstractFileStatus variableWordSubclass: #UniflexFileStatus

    instanceVariableNames:      ''
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Interface'
```

Summary

An instance of this class is a buffer which contains information about a file. The following questions are answered by file status information.

- Is the file a directory?
- When was the file last modified?
- What is the file descriptor?

The file status class is usually not used in isolation, but in conjunction with the system call class or with **FileStream**. As a result of the following code, three things happen — an instance of **UniflexFileStatus** is created, the buffer is filled, and the instance is returned:

```
OS status: aFileDescriptor
```

In the code above, **UniflexSystemCall** is referred to by the global variable **OS**. See the *4400 Series Assembly Language* manual, Section 4 System Calls, "status" for details about the contents of the buffer.

Instance Methods

accessing

buffer
The receiver is the buffer which holds the file status information.

fileSize
Answer the file size in bytes.

isDirectory
Answer whether the file represented by the receiver is a directory.

isReadable
Answer true if the file represented by the receiver is readable.

isWritable
Answer true if the file represented by the receiver is writable.

lastModified
Answer the time of the last modification to the file.

longDescription
Answer a String that contains a description of the receiver which looks like a line from a dir command.

comparing

= aFileStatus
Answer true if the fdn and device number are the same.

hash
Hash is reimplemented because = is implemented.

Class Methods

instance creation

new
Answer a new instance of the receiver.

Rationale

This class is a concrete subclass of **AbstractFileStatus** and uses the protocol framework established there. **UniflexFileStatus** serves as an interface to the operating system structure which holds file status information.

```
AimSystemCall variableSubclass: #UniflexSystemCall

    instanceVariableNames:        ''
    classVariableNames:           ''
    poolDictionaries:             ''
    category:                     'OS-Interface'
```

Summary

This is the concrete class used to interface to the operating system on the Tek 4404, 4405 or 4406. It defines instance creation and portable operations implementation for the UniFLEX operating system used by these workstations.

Instance Methods

execution

systemInvokeQuietly
Make a system call. Return success or failure of the system call, or nil if the primitive fails.

portable subtask operations

terminatedSubtaskExitCode
Answer the low byte of the status returned from the wait system call. This portion represents the value of the argument supplied by the exit system call causing termination. The high order bit of the portion indicates whether the terminated task has made a core dump. The receiver must be an instance representing a wait system call, which has been executed.

terminatedSubtaskExitInterrupt
Answer the high byte of the status returned from the wait system call. This portion represents the value of the signal causing termination. The receiver must be an instance representing a wait system call, which has been executed.

terminatedSubtaskID
Answer the ID returned from the wait system call. The receiver must be an instance representing a wait system call, which has been executed.

Class Methods

class initialization

initialize
Initialize the error message array used by UniflexSystemCall objects.

environment variables

argCount
Return the number of arguments used to invoke Smalltalk.

originalEnvironment
Return the environment used to invoke Smalltalk.

file names

backupFileName: aFileName
Answer a string which is a backup file name for the file aFileName.

isBackupFileName: aFileName
Does aFileName correspond to a name that is usually a backup file name?

general inquiries

abnormalTerminationCode
Answer the code for abnormal task termination.

asTime: osSeconds
Convert the operating system's notion of time to a Time. The operating system has corrected for time zone and daylight savings time. Add in the total seconds from Jan. 1, 1901 (the start of Smalltalk time) up to Jan. 1, 1980 (the start of Uniflex time).

brokenPipeInterrupt
Answer the interrupt number for the broken pipes interrupt.

deadChildInterrupt
Answer the interrupt number for the dead child interrupt.

defaultInterruptCode
Answer an operating system representation of the default interrupt action.

fileBufferSize
Return the preferred size of a buffer used for reading files.

fileStatusClass
Answer the class whose instances hold the file status returned by system calls that are instances of the receiver.

fontDirectory
Return the directory which contains font files. Each file contains a font in external font format.

getMachineName
Return the type of machine Smalltalk is running on.

ignoreInterruptCode
Answer an operating system representation of the ignore interrupt action.

isValid
Does this class represent the operating system running on this machine? (This method should return true only if the underlying operating system is Uniflex.)

maxFileNameSize
Answer the maximum number of characters permissible for file names.

maxOpenFiles
Answer the maximum number of simultaneously open files.

nonBlockingWait
Answer true if the wait instance creation method returns a non-blocking call, false otherwise. Uniflex has a blocking wait call.

priorityInterval
Answer the interval of valid priorities in ascending order for this operating system. Maximum values range from 0 to 25, zero being the highest and 25 being the lowest; however, this range is restricted for most users.

returnKeyCode
Answer the Smalltalk character value which should be assigned when the return key is pressed. The Uniflex operating system uses the CR convention.

smalltalkInitializationDirectory
Return the directory which contains initialization files. Each file contains Smalltalk readable data used during class initialization.

terminateInterrupt
Answer the interrupt number for the terminate interrupt. This interrupt can be caught.

terminateUnconditionallyInterrupt
Answer the interrupt number for the terminate unconditionally interrupt. This interrupt cannot be caught.

validPriority: aPriority
Answer the validity of this priority for this task.

portable directory operations

changeDirectory: aString
Change directory to the specified directory.

createDirectory: aString
Make aString, a full path name, be a new directory.

currentDirectoryName
Answer the complete path name of the current working directory.

nextFileName: directoryStream
Answer the next file name in directoryStream. Advance the directory stream beyond that name. Answer nil if none.

Directories are formatted into 16-byte entries, the last 14 of which contain the characters of the file names. Short names use one entry with zero fill if required. Long names use multiple consecutive entries all of which are marked in the high-order bit of their first character position. A zero byte (ignoring marks) is guaranteed.

removeDirectory: aString
Remove the directory named aString. The directory must be empty (i.e., contain only . and ..).

portable file operations

create: aString
Create a new file named aString. Answer a writeOnly fileDescriptor for the file.

duplicateFd: fileDescriptor
Return a new file descriptor that references the same open file as fileDescriptor.

duplicateFd: oldFileDescriptor **with:** newFileDescriptor
Cause newFileDescriptor to reference the same open file as oldFileDescriptor. If newFileDescriptor currently references an open file, that file is first closed.

existingName: fileName
Answer true if a file or directory exists by the name fileName, a String.

freeFileDescriptors
Answer the number of available file descriptors.

newPipe
Return an instance of Pipe.

open: aString
Open the file named aString. Answer a readWrite fileDescriptor for the file.

openForRead: aString
Open the file named aString. Answer a readOnly fileDescriptor for the file.

openForWrite: aString
Open the file named aString. Answer a writeOnly fileDescriptor for the file.

read: fileDescriptor **into:** aStringOrByteArray
Fill aStringOrByteArray with data from the file referred to by fileDescriptor. Answer the number of bytes read. Answer zero if at end.

read: fileDescriptor **into:** aStringOrByteArray **size:** count
Fill aStringOrByteArray with, at most, count data elements from the file referred to by fileDescriptor. Return the number of bytes read, or zero if at end.

rename: aFileName **as:** newFileName
Rename the file named aFileName to have the name newFileName. Create an error if aFileName does not exist; but not if newFileName exists.

seek: aFileDescriptor **to:** aFilePosition
Position the file represented by aFileDescriptor to aFilePosition bytes from its beginning.

shorten: fileDescriptor
Shorten a file to its current position.

size: fd
Return the count of available bytes from the file or pipe known by the file descriptor fd.

status: fd
Return a UniflexFileStatus for the file known by the file descriptor fd.

statusName: fileName
Answer a UniflexFileStatus for the file referred to by fileName, a String.

validFileDescriptor: fileDescriptor
Answer true if an open file with the specified file descriptor exists.

write: fileDescriptor **from:** aStringOrByteArray **size:** byteCount
Write byteCount bytes of data from aStringOrByteArray to the file referred to by fileDescriptor.

portable subtask operations

brokenPipesProcessWith: aBlock
Answer a process that monitors broken pipes. ABlock is executed after the receipt of each broken pipe signal.

executeUtility: aCommand **withArguments:** anOrderedCollection
Execute a binary program and return the entire results generated by the program as a string. No mechanism for input to the program is provided. Notify an error if the program cannot be executed or if the program terminates abnormally.

executeUtilityWithErrorMapping: aCommand
withArguments: anOrderedCollection
Execute a binary program and return an array of two strings. The first string contains the entire normal output generated by the program. The second string contains any error message output from the program. No mechanism for input to the executable program is provided. Notify an error if the program cannot be executed or if the program terminates abnormally.

forkShell
Fork an operating system shell with history. Type 'exit' to the shell to return to Smalltalk.

sendInterrupt: anInterruptID **to:** aTaskID
Send an 'interrupt' to a task by specifiying an interruptID and a taskID. Return true if the operation was successful, false otherwise. First check to see if aTaskID is a valid task ID.

setInterrupt: anInterruptID **to:** aSemaphoreOrParameter
Override the default action for an 'interrupt' by connecting it to a semaphore or some system specific parameter. If specified, the semaphore is posted on interrupt. After the interrupt is received, the interrupt must be reconnected or it will return to its default action. The exception to this is the DeadChildInterrupt, number 26.

setTaskPriority: priority
Set the priority of this task to a value which should be included in this class' priority interval.

shell
Fork an operating system shell with history. Type 'exit' to the shell to return to Smalltalk.

startSubtask: executeCall **withBlock:** childBlock
Start the subtask by spawning a child task. Then, the child task only evaluates the childBlock, and executes the executeCall to transform the child task into another program. If the executeCall fails, terminate the child task. The child task will inherit the priority of the Smalltalk task. Answer the spawned child task ID, nil if none.

terminate: aTaskID
Using an interrupt, attempt to terminate the task associated with the specified ID. This termination is requested in a manner which can be intercepted.

terminatedSubtasksProcessWith: aBlock
Answer a process that monitors spawned child tasks. ABlock is executed after the termination of each child task. The dead child signal is automatically reset by the operating system.

terminateUnconditionally: aTaskID
Terminate this task unconditionally.

system-environment variables

invocationArgCountAddress
Get the address of the argument count used in this Smalltalk.

system-files

chacc: fileName **mode:** permInteger
Check the accessibility of the file fileName.

chdir: directoryName
Change directory.

chown: fileName **to:** ownerID
Change owner of a fileName to ownerID.

chprm: fileName **to:** permissions
Change permissions for a file.

close: fileDescriptor
Close a file.

controlPty: fileDescriptor **command:** cmd **mode:** modeByte
Control, via the master end, the slave side of a pty. The fileDescriptor is a master mode pseudo-terminal file descriptor. The state of the channel is returned in D0. The argument cmd is a subfunction and modeByte is the argument to the subfunction.

create: fileName **mode:** modeBits
Create a new file.

createPty
Create a pseudo terminal master/slave device pair, known as a channel. The file descriptor for slave access is returned in D0 and the file descriptor for master access is returned in A0. Once the channel has been created, additional slave accesses may be created using UniflexSystemCall open:mode:. UniflexSystemCall ofstat:buffer: may be used to get information about the status of master and slave sides.

crtsd: newName **mode:** mode **addr:** addr
To create a directory, mode should be 8r04077 and addr should be 0.

defacc: permissions
Set the default permissions.

dup: fileDescriptor
Duplicate the file descriptor; open the file again.

dups: fileDescriptor **with:** specifiedDescriptor
Duplicate the file descriptor, specifying the file descriptor of the duplicated open file.

fcntl: fileDescriptor **function:** controlFunction
Change or interrogate the behavior of a file. The state of the modifiable behaviors is returned in D0.

controlFunction	Result
0	Get the state (1 = noblock, 2 = block).
2	Answer file descriptor of last file which sent INPUT READY signal, -1 if none.
3	Subsequent reads on this descriptor do *not* block. Error ENOINPUT is returned if no data and signal INPUT READY sent when data becomes available.
4	Subsequent reads do block.

filtime: fileName **to:** time
Set last modified time of file.

link: fileName **to:** linkName
Link a file to a link name.

ofstat: fileDescriptor **buffer:** buff
Get the status of an open file.

open: fileName **mode:** modeBits
Open a file.

status: fileName **buffer:** buf
Read the status of file fileName into the buffer buf. Buf should be aWordArray of size 11.

unlink: fileName
Unlink a file.

system-information

time: tbuff
Get the system time. Tbuff should be a WordArray of size 5.

ttime: tbuff
Get the current task's time. Tbuff should be a WordArray of size 8.

system-input output

read: fileDescriptor **buffer:** buff **nbytes:** numberOfBytes
Perform a read operation with the appropriately sized buffer.

seek: fileDescriptor **offset:** position **whence:** start
Change the position in the file.

truncate: fileDescriptor
Truncate the file at the current position.

ttyget: fileDescriptor **buffer:** ttybuff
Get the tty description of the specified open file. Ttybuff should be a WordArray of size 3.

ttynumber
Get the number of the calling task's terminal.

ttyset:fileDescriptor **buffer:** ttybuff
Set the tty information as described in ttyget.

update
Update the contents of system disks.

write: fileDescriptor **buffer:** buff **nbytes:** numberOfBytes
Write numberOfBytes from buff to a file.

system-resources

lrec: fileDescriptor **howmany:** count
Make an entry in system's locked record table.

rump: resourceName **operation:** resourceOperation
Create, destroy, enqueue or dequeue a named resource. Used to provide a mechanism for controlling access to physical resources. The resourceName must be no larger than 15 characters. ResourceOperation should be a SmallInteger , 1 representing enqueue, 2 representing dequeue, 3 representing create, and 4 representing destroy.

urec: fileDescriptor
Unlock a file record.

system-subtasks

alarm: seconds
Set alarm to go off in seconds. Return previous value of seconds in D0.

cpInt: interrupt **to:** aSemaphoreOrAddress
Set an interrupt from the operating system to signal a semaphore or tell the operating system upon interrupt to branch to an address. It may be used to restore the old interrupt address also. The old address (or semaphore) is returned in D0.

crPipe
Create a pipe.

exec: fileName **with:** argList
Execute a binary file specified by fileName. FileName is a character string. ArgList is an OrderedCollection of character strings. The task ID is returned in D0.

execute: program **withArguments:** argCollection
withEnvironment: environmentDictionary
Answer an instantiated instance of the exec system call with arguments and environment variables in the proper format. The exec call has not been invoked yet.

execve: pathName **withArgs:** argList **withEnv:** envList
Execute a binary file specified by pathName. PathName is a character string. ArgList is an OrderedCollection of character strings. EnvList is an OrderedCollection of environmental strings in the following format: environment name = the corresponding environment value string (no spaces around equal). For example, 'HOME=/public'.

If the executable program is /bin/shell or /bin/script, it will create a default environment if it is passed a null environment. The task ID is returned in D0.

exit: exitParam
Answer an instance of the exit system call. It has not been invoked yet.

fork
Answer an instance of the appropriate fork system call. It has not been invoked yet.

forkProcess
Create a new task. The new task receives a copy of the entire address space. Return the task's ID in D0.

getId
Get the running task's ID.

getuId
Get the actual user ID and the effective user ID.

setpr: priority
Set the priority.

setuId: userID
Set the actual user ID and the effective user ID.

spint: taskNumber **an:** interrupt
Send a task, specified by taskNumber (ID), an interrupt.

term: terminatingStatus
Terminate a task with an error indicating status (zero = no error).

vfork
Create a new child task. The new task does not receive a copy of the entire address space, instead the parent and child processes share memory. Return the child task ID in D0.

wait
Wait for a child or a program interrupt. D0 returns taskID and A0 returns termination status.

yieldCPU
Yield the CPU to tasks of equal priority.

Rationale

UniflexSystemCall is the primary interface between Smalltalk and the operating system on Tektronix 4400 series workstations. Smalltalk requires operating system services, the very least of which is being able to interact with the UniFLEX operating system file system. Additionally, there are many other functions/programs accessible via the operating system that should be usable from within Smalltalk.

Related Classes

AbstractSystemCall
AimSystemCall
Subtask

```
AbstractFileStatus subclass: #UTekFileStatus

    instanceVariableNames:      'buffer '
    classVariableNames:         ''
    poolDictionaries:           ''
    category:                   'OS-Interface'
```

Summary

An instance of this class contains a buffer holding information about a file. Information may be things like

- Is the file a directory?,
- Last time the file was modified, and
- File descriptor.

The file status class is not usually used in isolation, but in conjunction with the system call class or with **FileStream**. As a result of the following code, three things happen — an instance of **UTekFileStatus** is created, the buffer is filled, and the instance is returned:

```
OS status: aFileDescriptor.
```

In the code above, **UTekSystemCall** is referred to by the global variable **OS**.

Instance Variables

buffer <Stat>
: The C structure which contains the file information.

Instance Methods

accessing

buffer
: Answer the buffer which holds the file status information.

fileSize
: Answer the file size in bytes.

isDirectory
Answer true if the receiver is a directory.

isReadable
Answer true if the file represented by the receiver is readable.

isWritable
Answer true if the file represented by the receiver is writable.

lastModified
Answer the time of the last modification to the file.

comparing

= aFileStatus
The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file.

hash
Hash is reimplemented because = is implemented.

Class Methods

instance creation

new
Answer a new instance of the receiver containing an instance of the C structure class Stat.

Rationale

This class is a concrete subclass of **AbstractFileStatus** and uses the protocol framework established there. **UTekFileStatus** serves as an interface to the operating system C structure which holds file status information.

```
AimSystemCall variableSubclass: #UTekSystemCall

    instanceVariableNames:      ''
    classVariableNames:         'SystemCallKeywords '
    poolDictionaries:           ''
    category:                   'OS-Interface'
```

Summary

UTekSystemCall is the class used to interface to the operating system on Tektronix UTek workstations. It is normally accessed through the global variable **OS**. In practice, a specific instance creation message is sent to the **UTekSystemCall** in order to create a data object with the proper format, then the message **invoke** is sent to the new system call object to actually cause the execution of the system call, with success or failure being returned. A similar message, **value**, causes a notifier upon system call failure, and returns the system call return value upon success.

This class is responsible for implementing all the protocol defined in its abstract superclasses, **AimSystemCall** and **AbstractSystemCall**.

Instance Variables

Inherited Instance Variables

A0In <PointerArray>
: This inherited variable contains the argument(s), if any, to the system call.

D0In <Integer>
: This inherited variable contains the value of the opcode symbol in the SystemCallKeywords dictionary.

D0Out <Integer>
: This inherited variable contains the value returned from the system call.

D1Out <Integer>
: This inherited variable sometimes contains supplementary return value information.

errno <Integer>
: This inherited variable contains an error number upon system call failure.

operation <Symbol>
This inherited variable contains the symbolic name for the system call.

operationType <Symbol>
This inherited variable contains a message selector identifying the action to take upon receiving the #invoke or #value message.

All other inherited instance variables are ignored.

Class Variables

SystemCallKeywords <Dictionary>
Symbols corresponding to the system call names are associated with system call indices.

Inherited Class Variables

ErrorMessages <Array>
This inherited variable contains strings associated with error numbers.

Pool Dictionaries

Inherited Pool Dictionaries

ErrorConstants
This inherited pool contains symbolic names for error numbers.

OSConstants
This inherited pool contains symbolic names for commonly used numeric constants.

Instance Methods

initialize-release

operation: opcode
Set up a system call with no arguments. Set the operation to the proper code.

operation: opcode **with:** arg0
operation: opcode **with:** arg0 **with:** arg1
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2 **with:** arg3
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2 **with:** arg3 **with:** arg4
operation: opcode **with:** arg0 **with:** arg1 **with:** arg2 **with:** arg3 **with:** arg4 **with:** arg5
Set up the arguments for a system call. Set the operation to the proper code.

accessing

at: index **put:** argument
Place a system call argument in the desired stack position.

constants

systemCallKeywordFor: syscallIndex
Answer a String from the SystemCallKeywords dictionary that corresponds to syscallIndex.

systemCallValueFor: aSymbol
Answer a value from the SystemCallKeywords dictionary that corresponds to aSymbol. If aSymbol is not found, notify the user that it is a bad key.

errors

error: errorLabel
Report a system call error with a verbose explanation.

errorString
Return a string that is associated with the present error code. If there is no string available, return a String representation of the error code.

issueError
Issue a notifier with a string that identifies the failed UTekSystemCall.

execution

returnD1
Cause the system call primitive to return a value in D1Out.

signalInvoke
Perform signal/semaphore mapping.

systemInvokeQuietly
Make a system call. Return success or failure of the system call, or nil if the primitive fails.

value
Evaluate the system call represented by the receiver. Answer the return value for a successful call; create an error notifier otherwise.

valueIfError: aBlock
Evaluate the system call represented by the receiver. Evaluate aBlock if the system call resulted in an error. Return the result of the system call otherwise.

waitInvoke
Make a wait call. Return success or failure of that system call. Notify if the primitive failed. This method is necessary because wait and wait3 have an assembly code interface that differs from all other system calls.

operation type

signalOperation
The desired operation involves a signal/semaphore link.

waitOperation
The desired operation involves a wait or wait3 system call.

portable subtask operations

terminatedSubtaskExitCode
Return a portion of the status returned from the wait system call. This portion represents the value of the argument supplied by the exit system call causing termination. The high order bit of the portion indicates whether the terminated task has made a core dump. The receiver must be an instance representing a wait system call, which has been executed.

terminatedSubtaskExitInterrupt
Return a portion of the status returned from the wait system call. This portion represents the value of the signal causing termination. The receiver must be an instance representing a wait system call, which has been executed.

printing

printOn: function
Print a representation of the system call similar to that used in the UTek manual.

Class Methods

class initialization

firstTime
Initialize a brand new image under Smalltalk. This method assumes that Disk and SourceFiles are initialized.

initialize
Read in the constants and messages needed for UTek System Calls.

initializeErrorMessages
Load the ErrorConstants pool. Load the class array ErrorMessages using the UTek "msghlp" utility.

constants

constant: aString
Return the value associated with aString.

keysAtValue: val
Return a set of symbols that are associated with the value val.

systemCallKeywordFor: syscallIndex
Return a Symbol from the SystemCallKeywords dictionary that corresponds to syscallIndex.

systemCallValueFor: aSymbol
Return a value from the SystemCallKeywords dictionary that corresponds to aSymbol. If aSymbol is not found, notify the user that it is a bad key.

environment variables

argCount
Return the number of arguments used to invoke Smalltalk.

originalEnvironment
Return the environment used to invoke Smalltalk.

file names

backupFileName: aFileName
Answer a string which is a backup file name for the file aFileName.

isBackupFileName: aFileName
Does aFileName correspond to a name that is usually a backup file name?

general inquiries

abnormalTerminationCode
Return the code for abnormal task termination.

asTime: osSeconds
Convert the operating system's notion of time to a Time. Add in the total seconds from Jan. 1, 1901 (the start of Smalltalk time) up to Jan. 1, 1970 (the start of UTek time). Add a correction value for time zone and daylight savings time.

brokenPipeInterrupt
Return the interrupt number for the broken pipes interrupt.

deadChildInterrupt
Return the interrupt number for the dead child interrupt.

defaultInterruptCode
Answer the code that will restore the default interrupt action.

fileBufferSize
Return the preferred size of a buffer used for reading files.

fileStatusClass
Answer the class of objects returned from system calls that return file status.

fontDirectory
Return the directory which contains font files. Each file contains a font in external font format.

getMachineName
Return the type of machine Smalltalk is running on.

ignoreInterruptCode
Answer the code that will cause interrupts to be ignored.

isValid
Does this class represent the operating system running on this machine? (This method should return true only if the underlying operating system is UTek.)

maxFileNameSize
Answer the maximum number of characters permissible for file names.

maxOpenFiles
Answer the maximum number of simultaneously open files.

nonBlockingWait
Does the UTekSystemCall #wait method return immediately rather than blocking?

priorityInterval
Return the interval of valid priorities in order of descending priority for this task and effective user.

returnKeyCode
Answer Smalltalk character value which should be assigned when the return key is pressed. UTek returns a linefeed.

smalltalkInitializationDirectory
Return the directory which contains initialization files. Each file contains Smalltalk readable data used during class initialization.

terminateInterrupt
Return the interrupt number for the terminate interrupt. This interrupt may be caught.

terminateUnconditionallyInterrupt
Return the interrupt number for an unconditional interrupt. This interrupt may not be caught.

validPriority: aPriority
Is aPriority a valid priority for this task and user?

portable directory operations

changeDirectory: aDirectoryName
Change the current directory to the specified directory.

createDirectory: directoryName
Create a new directory with the name directoryName.

currentDirectoryName
Return a String with the name of the current working directory.

nextFileName: directoryStream
Return the next existing file name in directoryStream. Advance the directory stream beyond that name. Answer nil if none. UTek directories have the following variant structure:

```
struct direct {
   u_long d_ino;           /*inode number for the file*/
   short  d_reclen;        /*length of this record, dword padded*/
   short  d_namlen;        /*length of the file name*/
   char   d_name[d_namlen+1]};     /*file name itself*/
```

To tell if a directory entry is unused, one must look at the previous entry to see if d_reclen is bigger than would be expected. This is not terribly easy to do with a Stream on variant records, so we take the coward's way out: get the name of the file, see if it exists, and either return the validated name, or go get the next one.

removeDirectory: directoryName
Remove the directory named directoryName.

portable file operations

create: fileName
Create a new file named fileName. Answer a writeOnly fileDescriptor for the file.

duplicateFd: fileDescriptor
Return a new file descriptor that references the same open file as fileDescriptor.

duplicateFd: oldFileID **with:** newFileID
Duplicate the existing file descriptor oldFileID, to the specified new file descriptor newFileID. If newFileID already existed, it is first closed. The old and new descriptors share an open file. The new file descriptor is returned.

existingName: fileName
Answer true if a file or directory exists by the name fileName.

freeFileDescriptors
Answer the number of available file descriptors.

newPipe
Return an instance of Pipe.

open: fileName
Open the file named fileName. Answer a readWrite fileDescriptor for the file.

openForRead: fileName
Open the file named fileName. Answer a readOnly fileDescriptor for the file.

openForWrite: fileName
Open the file named fileName. Answer a writeOnly fileDescriptor for the file.

read: fileDescriptor **into:** aStringOrByteArray
Fill aStringOrByteArray with data from the open file known by fileDescriptor. Return the number of bytes read, or zero if at end.

read: fileDescriptor **into:** aStringOrByteArray **size:** count
Fill aStringOrByteArray with, at most, count data elements from the file referred to by fileDescriptor. Return the number of bytes read, or zero if at end.

rename: fileName **as:** newFileName
Rename the file named fileName to have the name newFileName. Create an error if fileName does not exist, but not if newFileName exists.

seek: aFileDescriptor **to:** aFilePosition
Position the file known by aFileDescriptor to aFilePosition bytes from its beginning. Return the resulting position.

shorten: fileDescriptor
Shorten a file to its current position.

status: fd
Return a FileStatus for the file known by the file descriptor fd.

statusName: fileName
Return a FileStatus for the file named fileName.

validFileDescriptor: fd
Answer true if an open file with the specified file descriptor exists.

write: fileDescriptor **from:** aStringOrByteArray **size:** byteCount
Write byteCount bytes of data from aStringOrByteArray to the file known by fileDescriptor. Return the actual number of bytes written.

portable subtask operations

brokenPipesProcessWith: aBlock
Return a process that monitors broken pipes. ABlock is executed after the receipt of each broken pipe signal.

defaultInterrupt: anInterruptID
Set the specified interrupt to its default action. The previous action is returned.

execute: program **withArguments:** argCollection **withEnvironment:** envDictionary
Answer an instance of the exec system call that has not yet been invoked.

forkShell
Set up the display and signal environment for terminal emulation, and turn it over to a forked shell Subtask. Block on the Subtask until it terminates, then restore the display and signal environment for Smalltalk.

ignoreInterrupt: interruptID
Set the specified interrupt to be ignored. The previous action is returned.

sendInterrupt: interruptID **to:** taskID
Send the interrupt known by interruptID to the task known by taskID. Return true if the operation was successful, false otherwise. First check to see if taskID is a valid task ID.

setInterrupt: interruptID **to:** aSemaphoreOrAddress
Override the default action for the interrupt known by interruptID by connecting it to aSemaphore or the address of a subroutine. The semaphore is posted on interrupt. The old semaphore (or address) is returned.

setTaskPriority: priority
Set the priority of Smalltalk to the value priority.

shell
Set up the display and signal environment for terminal emulation. Either suspend Smalltalk (for shells that support it) or fork a new shell. Restore the display and signal environment for Smalltalk upon resumption.

startSubtask: execCall **withBlock:** childBlock
Fork a copy of Smalltalk. In the child copy, execute childBlock and invoke execCall, which must be an instantiated 'exec' system call. If execCall returns, there is an error: terminate the child task. Meanwhile, the parent task returns the child task ID.

terminate: taskID
Using an interrupt, attempt to terminate the task known by taskID. This termination is requested in a manner which can be intercepted.

terminatedSubtasksProcessWith: aBlock
Return a Process that moniters spawned child tasks. ABlock is executed after the termination of each child task. The dead child signal is automatically reset by the operating system.

terminateUnconditionally: aTaskID
Terminate this task unconditionally.

wait
Answer an instance of the wait system call that has not yet been invoked.

system-files

access: pathName **mode:** modeBits
Access returns the value 0 in D0 if the file pathName is accessible for reading, writing, or executing according to the modeBits.

chdir: pathName
Change the current working directory to pathName.

chmod: pathName **mode:** modeBits
Change the file pathName to have a mode described by modeBits.

close: fileID
Delete the file referenced by the file descriptor, fileID, from the reference table. If this is the last reference to the underlying object, it will be deactivated.

dup2: oldFileID **newfd:** newFileID
Duplicate the existing file descriptor oldFileID, to the specified new file descriptor newFileID. If newFileID already existed, it is first closed. The old and new descriptors share an open file. The new file descriptor is returned in D0.

dup: oldFileID
Duplicate the existing file descriptor oldFileID, by creating a new file descriptor. The old and new descriptors share an open file. The new file descriptor is returned in D0.

fchmod: fileID **mode:** modeBits
Change the file denoted by fileID to have a mode described by modeBits.

fcntl: fileID **cmd:** commandID **arg:** commandArg

Fcntl is used to control various aspects of the open file known by the descriptor fileID, depending on the Integer arguments, commandID and commandArg. An Integer value is returned in D0. The following table describes the various actions of fcntl (X means don't care or undefined).

commandID	commandArg	D0Out	Action
F_DUPFD	an fd	new fd	return a new fd >= commandArg
F_GETFD	X	flag	return close-on-exec flag
F_SETFD	0 or 1	X	set close-on-exec flag to commandArg
F_GETFL	X	status flags	return current status flags
F_SETFL	status flags	X	set status flags
F_GETOWN	X	pg negated	return process group
F_SETOWN	pg negated	X	set process group to pg
F_SETOWN	pid	X	set process group to that of process pid

Status flags are defined as a bitOr: of the following values:

FNDELAY	FileID is in non-blocking mode. If a read or write would block, the call fails with the error code EWOULDBLOCK.
FAPPEND	Writes to fileID are appended to the end of the file.
FASYNC	Send SIGIO to the process group when I/O is possible.

flock: fileID **operation:** operationID

Flock controls advisory locks that cooperating processes may associate with files. Locks can be applied either exclusively or shared, and may be set non-blocking. OperationID is a set of bit flags used to determine the type of lock that will be applied to the open file descriptor fileID.

fstat: fileID **buf:** statStructure

Fstat returns information in statStructure about the open file known by the descriptor fileID.

link: oldPathName **path2:** newPathName

Create a new directory entry newPathName for the file oldPathName. Both paths must be on the same file system.

lstat: pathName **buf:** statStructure
Lstat returns information in statStructure about the file or symbolic link pathName.

mkdir: directoryName **mode:** modeBits
Mkdir creates a new directory directoryName with the mode being modeBits combined with the current mode mask.

open: pathName **flags:** options **mode:** modeBits
Open the file pathName under the control of options. If the file does not exist and options specify creation, set the new file's mode to modeBits, in combination with the current mode mask. A file descriptor by which the open file is known is returned in D0.

readlink: symbolicLinkPathName **buf:** linkByteData **bufsiz:** linkSize
Readlink returns in linkByteData the contents of the symbolic link symbolicLinkPathName. LinkByteData must be at least linkSize. The number of characters read is returned in D0.

rename: oldFileName **to:** newFileName
Rename causes the file oldFileName to be known as newFileName.

rmdir: directoryPathName
Rmdir removes the directory known as directoryPathName.

stat: pathName **buf:** statStructure
Stat returns information in statStructure about the file pathName.

symlink: oldPathName **path2:** newPathName
Symlink creates a symbolic link to oldPathName in a file named newPathName.

umask: newUmask
Umask sets the file creation mode mask for Smalltalk to newUmask. The previous mode mask is returned in D0.

unlink: pathName
Unlink removes the reference to pathName from its directory. The file will not go away if there are other links to it or it is open in any process.

utimes: pathName **tvp:** timevalStructurePair
The last access and last modified times, respectively, for the file pathName are returned in timevalStructurePair.

system-information

getdtablesize
Getdtablesize returns the number of files that may be open at any given time in D0, which is the size of the descriptor table. This value exists in a constant and is obtained quickly via the expression:

UTekSystemCall constant: 'NOFILE'.

getegid
Getegid returns the actual group ID of the user running Smalltalk in D0 and the effective group ID of the user running Smalltalk in D1. This may be different than the actual group ID of the user running Smalltalk if Smalltalk has the set-group-ID bit set.

geteuid
Geteuid returns the actual user ID of the user running Smalltalk in D0 and the effective user ID of the user running Smalltalk in D1. This may be different than the actual user ID of the user running Smalltalk if Smalltalk has the set-user-ID bit set.

getgid
Getgid returns the actual group ID of the user running Smalltalk in D0 and the effective group ID of the user running Smalltalk in D1. This may be different than the actual group ID of the user running Smalltalk if Smalltalk has the set-group-ID bit set.

gethostname: hostName **namelen:** hostNameSize
Gethostname places this machine's name in hostName, which must contain room for at least hostNameSize characters.

getitimer: timerType **value:** timerStructure
Getitimer returns in timerStructure information for one of three interval timers, determined by timerType.

getpagesize
Getpagesize returns the number of bytes in a Memory Management Unit page in D0.

getrlimit: resourceID **rlp:** rlimitStructure
Return in rlimitStructure the resource limits imposed on the Smalltalk process and its children.

gettimeofday: timevalStructure **tzp:** timezoneStructure
Return the system's notion of the current Greenwich Mean Time and the local time zone in timevalStructure and timezoneStructure.

getuid
Getuid returns the actual user ID of the user running Smalltalk in D0 and the effective user ID of the user running Smalltalk in D1. This may be different than the actual user ID of the user running Smalltalk if Smalltalk has the set-user-ID bit set.

mstat: memoryType **addr:** startAddress **len:** length **vec:** processString
Return in processString information describing the process clusters of a particular memoryType beginning at startAddress and continuing for length bytes.

profil: tallyByteArray **bufsiz:** tallySize **offset:** pcStart **scale:** granularity
Profil allows execution time profiling by examining the user program counter each clock interval. Offset is subtracted from the PC, multiplied by granularity, and tallied in tallyByteArray if the result is within tallySize.

setitimer: timerType **value:** timerStructure **ovalue:** oldTimerStructure
Setitimer sets one of three interval timers, determined by timerType, to the value specified in timerStructure. The previous value for that timer is returned in oldTimerStructure.

setregid: realGroupID **egid:** effectiveGroupID
Setregid sets the real and effective group IDs of Smalltalk to realGroupID and effectiveGroupID, respectively. Only root may change the real group ID, others may only set the effective group ID to the real group ID.

setreuid: realUserID **euid:** effectiveUserID
Setreuid sets the real and effective user IDs of Smalltalk to realUserID and effectiveUserID, respectively. Only root may change the real user ID, others may only set the effective user ID to the real user ID.

setrlimit: resourceID **rlp:** rlimitStructure
Setrlimit changes the resource limits for Smalltalk and its children to the limits described in rlimitStructure.

uname: utsnameStructure
Uname returns in utsnameStructure information identifying the current operating system.

system-input output

fsync: fileID
Fsync causes an open file descriptor fileID, to have a disk representation consistent with its memory representation.

ftruncate: fileID **length:** truncatedSize

Ftruncate causes a file that is open for writing known by the descriptor fileID to be reduced to a given size, if it is currently larger than that size. Truncated data are lost.

ioctl: fileID **request:** commandID **argp:** argByteArray

Set or get the operating characteristics of the open device file known by the descriptor fileID according to commandID and argByteArray. CommandID is an Integer that both identifies the request and describes argByteArray:

```
union commandID {
    int command;
    struct field {
        short   in:1;           /*set if argByteArray gets input*/
        short   out:1;          /*set if argByteArray gets output*/
        short   unused:7;
        short   len:7;          /*length of argByteArray, max 128*/
        short   request}}       /*ioctl request code*/
```

(The fields in commandID are built properly by the "OS constant: requestString" expression.) Ioctl requests are device dependent. The following table links devices, available requests, and UTek manual pages from Command Reference, Volume 2, Section 4:

Device	Manual	Requests
ethernet	ARP, ILAN, LNA, LO, NETWORKING	Manipulate network interfaces.
parallel port	HC	Set/clear CRM/RCSM modes.
tape	MTIO	Control operation of raw cartridge tape device.
RS-232 port	RSA	Control operation of RS-232 port.
terminals	TTY	Control operation of console, tty port, and pseudo-ttys.

Terminals and RS-232 ports are the devices most often sent ioctl requests. The following requests are available for controlling these devices:

Request	Action
TIOCSETD, TIOCGETD	Set/get line discipline: old, new, or network.
TIOCSPGRP, TIOCGPGRP	Set/get process group for current process.
TIOCMODS, TIOCMODG	Set/get RS-232 port control lines.
TIOCSETP, TIOCGETP	Set/get parameters as in stty/gtty.
TIOCSETC, TIOCGETC	Set/get special characters.
TIOCFLUSH	Flush buffers.
TIOCSTI	Simulate terminal input.
TIOCPKT	Set/clear pseudo-tty packet.

. . . and many, many more. See TTY(4) for details.

lseek: fileID **offset:** bytes **whence:** offsetType
Lseek moves the file pointer associated with the descriptor fileID by bytes positions, and returns the new value of the file pointer in D0. OffsetType specifies absolute, incremental, or extending movement.

read: fileID **buf:** byteData **nbytes:** byteDataSize
Read data from the file known by the descriptor fileID into byteData, which must be of at least byteDataSize. The number of bytes actually read is returned in D0.

readv: fileID **iov:** bufferArray **iovcnt:** bufferArraySize
Read data from the file known by the descriptor fileID into the buffers referenced in bufferArray, which must be of at least bufferArraySize. Readv may not be used on raw devices or across a network. The number of bytes actually read is returned in D0.

select: fileCount **readfd:** readFiles **writefd:** writeFiles **exceptfd:** pendingFiles **timeout:** timevalStructure
Select examines the open files corresponding to the bit masks readFiles, writeFiles, and pendingFiles to see if they are ready for reading, writing, or have exceptional conditions pending, respectively. At most, fileCount descriptors are examined. ReadFiles, writeFiles, and pendingFiles then return a bit mask corresponding to those descriptors that were found ready, with the total number of ready descriptors returned in D0. If timevalStructure is zero, select blocks until all descriptors are ready, otherwise timevalStructure indicates how long to wait for ready descriptors.

sync
Sync causes all in-core information that should be on disk to be written out.

truncate: pathName **length:** truncatedSize
Truncate causes the file pathName to be reduced to a given size, if it is currently larger than that size. Truncated data are lost.

write: fileID **buf:** byteData **nbytes:** byteDataSize
Write writes byteDataSize bytes from the buffer byteData into the open file known by descriptor fileID. The number of bytes actually written is returned in D0.

writev: fileID **iov:** bufferArray **iovcnt:** bufferArraySize
Writev writes the number of buffers specified by bufferArraySize into the open file known by descriptor fileID These buffers are described iovecStructures contained in bufferArray. The number of bytes actually written is returned in D0.

system-sockets

accept: socketID **addr:** sockaddrStructure **addrlen:** sockaddrSize
Accept a connection on a socket by creating a new socket. The new socket has the same properties as the socket denoted by socketID. This operation follows blocking/nonblocking protocol. The length argument (a 4-byte quantity) initially contains a pointer to the size of the socket address buffer, and returns the new size of the buffer. The descriptor of the new socket is returned in D0.

bind: socketID **name:** sockaddrStructure **namelen:** sockaddrSize
Bind a name to a socket. After use, this socket must be deleted by the user with the unlink call. The socket, denoted by socketID, is bound to a name described by sockaddrStructure of length sockaddrSize.

connect: socketID **name:** sockaddrStructure **namelen:** sockaddrSize
Either specify the peer to which datagrams are sent or initiate a connection on another socket, socketID. The length of the socket name is specified by sockaddrSize.

getpeername: socketID **name:** sockaddrStructure **namelen:** sockaddrSize
Getpeername returns the name of the peer connected to socket socketID in sockaddrStructure. SockaddrSize is a pointer to the actual size of sockaddrStructure, which returns the actual size of the returned sockaddrStructure.

getsockname: socketID **name:** sockaddrStructure **namelen:** sockaddrSize
Return in sockaddrStructure the name of the socket socketID. SockaddrStructure must be of size sockaddrSize, and the size of the returned name is returned in sockaddrSize.

getsockopt: socketID **level:** optionType **optname:** optionID **optval:** optionString **optlen:** optionSize
Return the options of optionType and optionID associated with the socket socketID in optionString. OptionSize is the length allocated for optionString, and is modified to indicate the actual size of the returned optionString.

listen: socketID **backlog:** maximumQueueLength
Establish a backlog queue of maximumQueueLength for the socket socketID.

recv: socketID **buf:** byteData **len:** byteDataSize **flags:** options
Recv returns a message from the connected socket socketID in byteData, which must be of at least byteDataSize. The number of bytes actually received is returned in D0.

recvfrom: socketID **buf:** byteData **len:** byteDataSize **flags:** options **from:** sockaddrStructure **fromlen:** sockaddrSize
Recvfrom returns a message from the socket socketID in byteData, which must be of at least byteDataSize. If sockaddrStructure is zero, no source address is received, otherwise sockaddrStructure returns the source address. SockaddrStructure must be at least sockaddrSize bytes. The number of bytes actually received is returned in D0.

recvmsg: socketID **msg:** msghdrStructure **flags:** options
Recvmsg returns a message from the socket socketID as directed by the msghdrStructure, which locates the sockaddr structure, output buffers, and access rights to be used. The number of bytes actually received is returned in D0.

send: socketID **msg:** byteData **len:** byteDataSize **flags:** options
Send transmits the message byteData of length byteDataSize to the connected socket socketID, as controlled by options. The number of characters actually sent is returned in D0.

sendmsg: socketID **msg:** msghdrStructure **flags:** options
Sendmsg transmits the message described by msghdrStructure to the socket socketID, as controlled by options. The number of characters actually sent is returned in D0.

sendto: socketID **msg:** byteData **len:** byteDataSize **flags:** options **to:** sockaddrStructure **tolen:** sockaddrSize
Sendto transmits the message byteData of length byteDataSize to the socket socketID described by sockaddrStructure, as controlled by options. The number of characters actually sent is returned in D0.

setsockopt: socketID **level:** optionType **optname:** optionID **optval:** optionString **optlen:** optionSize
Set the options of optionType and optionID associated with the socket socketID to the values described in optionString, which is of size OptionSize.

shutdown: socketID **how:** endID
Shutdown causes all or part of a full-duplex connection with socket socketID to be closed. The part disconnected is described by endID.

socket: addressType **type:** socketType **protocol:** protocol
Socket creates a socket of socketType using the address format addressType and the given protocol. A new socket descriptor is returned in D0.

socketpair: domain **type:** socketType **protocol:** protocol **sv:** socketIDs
Socketpair creates a pair of connected sockets of socketType in the given domain using the given protocol. A pair of new socket descriptors is returned in socketIDs.

system-subtasks

execve: pathName **argv:** arguments **envp:** environmentVariables
Transform Smalltalk into a new UTek process. The file specified by pathName is either an executable object file, or a data file for an interpreter. Arguments is an array of null-terminated strings, the first of which must be the name of the executable program, pathName. EnvironmentVariables is an array of null-terminated strings, each of which is in the form "name=value". There is no return from this system call.

exit: status
Terminate Smalltalk with the given status. All descriptors are closed, the parent process may be notified of the status throught the wait system call, and existing child processes have their parent ID changed to 1. There is no return from this system call.

fork
Fork causes a copy of the current Smalltalk process to be created. The parent finds the new task ID in D0 and the value 0 in D1. The new task finds the parent task ID in D0 and the value 1 in D1.

getgroups: maxEntries **gidset:** groupSet
Getgroups stores the group access list for the Smalltalk process in the groupSet array, which contains at least maxEntries.

getpgrp: processID
Getpgrp returns the process group of the process processID, which, if zero, specifies this Smalltalk. The group ID is returned in D0.

getpid
Return the process ID of Smalltalk in D0.

getppid
Return the process ID of Smalltalk in D0. Return the process ID of Smalltalk's parent in D1.

getpriority: priorityType **who:** identifier
Return the scheduling priority of a process, process group, or user as determined by priorityType. Identifier is either a process ID, process group ID, or user ID, as appropriate. The priority is returned in D0.

getrusage: whoFlag **rusage:** rusageStructure
Return in rusageStructure information about resource usage of Smalltalk or its terminated children, as indicated by whoFlag.

kill: processID **sig:** signalID
Send the process known by processID the signal signalID.

kill: processID **sig:** signalID **arg:** signalArgument
Send the process known by processID the signal signalID with the argument signalArgument.

killpg: processGroupID **sig:** signalID
Send the process group known by processGroupID the signal signalID.

pipe
Pipe creates an interprocess communication channel. The read end of the pipe is known by the descriptor in D0, while the write end of the pipe is known by the descriptor in D1.

ptrace: actionCode **pid:** processID **addr:** processAddr **data:** processData
Ptrace allows interactive control of a child process processID. The actionCode argument controls interpretation of the processAddr and processData arguments.

setpgrp: processID **pgrp:** groupID
Setpgrp sets the process group of the process processID to groupID.

setpriority: priorityType **who:** identifier **prio:** newPriority
Set the scheduling priority of a process, process group or user as determined by priorityType to the value newPriority. Identifier is either a process ID, process group ID, or user ID, as appropriate.

sigblock: blockedSignalsMask
Sigblock causes signals specified in blockedSignalsMask to be added to the set of signals currently blocked. A mask representing the signals previously blocked is returned in D0.

sigpause: blockedSignalsMask
Sigpause causes signals specified in blockedSignalsMask to become the set of signals currently blocked. It then waits for a signal to arrive, finally restoring the previous blocked signal mask.

sigsetmask: blockedSignalsMask
Sigmask causes signals specified in blockedSignalsMask to become the set of signals currently blocked. A mask representing the signals previously blocked is returned in D0.

wait3: waitStructure **options:** blocking **rusage:** rusageStructure
Wait3 checks the status of child processes, optionally without suspending, as controlled by blocking. Upon return from wait, waitStructure contains both the exit code and the termination status of the child that died, unless waitStructure was set to zero before the call, in which case no status is returned. If rusageStructure is non-zero, information concerning the resource usage of terminated children is returned in rusageStructure. The process ID of the stopped or terminated child process is returned in D0.

wait: waitStructure
Wait suspends Smalltalk until it receives a signal or one of its children dies. Upon return from wait, waitStructure contains both the exit code and the termination status of the child that died, unless waitStructure was set to zero before the call, in which case no status is returned. The process ID of the stopped or terminated child process is returned in D0.

Rationale

UTekSystemCall is the primary interface between Smalltalk and the operating system. Smalltalk requires operating system services, the very least of which is being able to read operating system files. Additionally, there are many other functions/programs accessible via the operating system that should be usable from within Smalltalk.

Background

Essential operating system services are defined in protocol found in either **AbstractSystemCall** or **AimSystemCall**. Those two classes establish the pattern of protocol to be implemented or overridden by methods in the subclasses for specific operating systems. Examples of essential services are **open:** and **closeFile:** to open and close files in the operating system; **read:into:** and **write:from:size** to read and write files; **changeDirectory:** to move to a different directory; and calls to **fork** and **execve** needed for subtasking. In addition to implementing or overriding protocol in its two immediate superclasses, **UTekSystemCall** contains specific protocol needed for the UTek operating system interface and UTek-specific services such as network access.

System Calls

Often the intended outcome of a system call is the side effect of the call (e.g., writing to a file) rather than what the call returns. In the UTek operating system, system calls are generally used by C functions that directly invoke UTek system operations. In Smalltalk, however, the UTek system operations are invoked via Smalltalk primitives. There are three types of system interface operations:

- display operations (primitive method **displayInvoke**),
- semaphore or signal operations (primitive method **signalInvoke**), and
- system and wait operations (primitive method **systemInvokeQuietly**).

When one of the three types of calls is made, the communication between Smalltalk and the operating system is accomplished through one of the three primitives — the second and third types of system interface operations are implemented in **UTekSystemCall**. These are system calls which are found in Section 2 of the manual *UTek Command Reference*. Display operations are implemented in **AimSystemCall**.

You should read the *intro(2)* section in *Utek Command Reference* for additional information about system call error messages and terminology.

Discussion

Most system calls will be accomplished by methods found in the message categories with "portable" in their name, for example, *portable directory operations*. Under those categories are the message selectors that will fulfill the most common system call requirements.

UTekSystemCall also provides non-portable methods under "system-" categories for every system call in Section 2 of the *UTek Command Reference* except

- calls that could destroy the running Smalltalk process/task (e.g., memory management calls), and
- system calls which are only accessible to root.

Before using these non-portable calls, the documentation of the call in Section 2 of *UTek Command Reference* should be reviewed.

Naming Convention

The naming convention for the message selectors under "system-" categories is as follows. The first keyword is the name of the UTek system call. The first argument is the first argument expected by the system call. Succeeding keywords and arguments name and supply, respectively, the remaining arguments to the system call. For example, the C specification `newfd = dup2(oldfd,newfd)` (*dup2(2)* in *UTek Command Reference*) is implemented as this **UTekSystemCall class** message selector:

dup2: oldFileID **newfd:** newFileID

dup2 is the system call name.
oldFileID is the first argument.
newfd: names the second argument, newFileID.

Class Protocol — *Portable* and *System-* Methods

Wherever possible, the "portable" operations should be used to ensure operating system independence. In images released for the UTek operating system, the global variable **OS** is set to **UTekSystemCall.** **OS** should be used in code (instead of the name of your system call class) to insure the portability of the code.

The class message categories beginning with "portable" should be checked first when you need to make a system call. Those categories will be the ones you use most frequently. The "portable" methods often send messages found under "system-" categories, but most of the "system-" methods can be thought of as *private.*

For example, suppose you want to duplicate a file descriptor. The *portable file operations* category contains the method **duplicateFd:**. This is the message that you should use. It works by calling **dup:**, which is found under *system-files*. The message **dup:** is sent to the global variable **OS**. If the system call fails, a notifier is displayed; otherwise, **dup:** returns the new file descriptor.

There is no way to tell from looking at the **dup:** method that the preferred message for your purpose is **duplicateFd:** — just follow the rule of thumb to look under "portable" first for the message you need. If you use a non-portable message selector, it will work but may not be as "user-friendly" as the portable one — **dup:** doesn't display the notifier that **duplicateFd:** displays when appropriate. In addition, moving your code to other Tektronix Smalltalk images will be much easier if you use the "portable" operations.

Portable directory operations allow you to manipulate directories, such as changing the current directory, enumerating the files in a directory, and creating or removing a directory.

Portable file operations allow you to perform common operations on files, such as creating, checking for existence, various file descriptor operations, renaming, opening, reading, positioning, creating pipes, writing, and obtaining file status information.

Portable subtask operations allow you to create new processes, send interrupts to processes, and change the response to interrupts.

The protocol is arranged with all the direct-map system calls together under "system-". "Direct-map" methods are those described under the preceding section called "Naming Convention". If no portable message exists for what you need to do, for example, socket operations or *ioctl*, the direct system call message should be used.

System-files methods are system calls that deal with testing, opening, closing, status, names, and descriptors of files.

System-information methods are system calls that return information from the system, such as user and group ID, host name, and timers. They are primarily concerned with returning information, as opposed to taking some action.

System-input output methods are system calls that read, write, seek, and deal with input and output in other ways.

System-sockets methods are system calls that are concerned with creating, connecting, receiving, and sending data on sockets.

System-subtasks methods are system calls that are concerned with UTek processes — creating, interrupting (signaling), and process ID.

Class Protocol — Other Message Categories

Class initialization methods initialize the class variables and pool dictionaries, if necessary. The **firstTime** method initializes the Smalltalk image for a specific operating system.

Certain operating system information is available (without making a system call) using class protocol for *constants, file names,* and *general inquiries.* The *environment variables* protocol supplies information about the environment at the time Smalltalk was invoked.

Instance Protocol

> **Note:** The only instance messages the typical user would send to an instantiated **UTekSysytemCall** are **value** or **invoke** (inherited from **AimSystemCall**).
> Remember that **invoke** does not perform error handling.

Initialize-release methods set up the instance variables and operation type of a system call, and set up arguments, if any, to be passed to the call.

The *accessing* method **at:put:** places an argument to the call in the position specified.

Constants methods return symbols or corresponding values, as specified, from the **SystemCallKeywords** dictionary. Superclass *constants* methods return information from the **OSConstants** and **ErrorConstants** pools.

Errors methods provide several formats of error messages.

Execution methods set instance variable **D1Out** to true, provide methods to execute a system call with error handling, and invoke the Smalltalk primitives for signal and system operations.

The *operation type* methods set instance variable **operationType** to #signalInvoke in order to invoke a signal/semaphore primitive or to #waitInvoke to make a *wait* or *wait3* system call. The superclass implements the methods **displayOperation** and **systemOperation.**

Portable subtask operations is also found under class protocol. Class protocol named "portable" causes creation of an instance of **UTekSysytemCall**; then the system call is made and the return value causes an error notifier if the call failed. You will see that instance *portable subtask operations* do not make a system call — they supply information available after a *wait* system call has executed.

The *printing* method **printOn:** prints the system call and its arguments using a format similar to the C specification for the system call.

Examples

The following example illustrates accessing the shell environment.

```
OS originalEnvironment at: #HOME.
```

If you execute the preceding code in a workspace with "printIt", the string representing your home directory will be printed. The following code will change your current directory to the directory above your current one.

```
OS changeDirectory: '..'.
OS currentDirectoryName.
```

If you execute the preceding code in a workspace with "printIt", the string representing the directory will be printed. The string will have the trailing "/" separator.

You can return to your home directory this way:

```
OS changeDirectory: (OS originalEnvironment at: #HOME).
```

The following code can be executed in a workspace. It illustrates various system calls for file operations found in the class message category *portable file operations.*

```
fd ← OS open: 'foo'.
OS write: fd from: 'Hi, folks!' size: 10.
OS seek: fd to: 0.
answer ← String new: 10.
OS read: fd into: answer.
OS closeFile: fd.
Transcript cr; show: answer.
```

This is what occurs in the example above. A file named 'foo' is created on the disk and its file descriptor is assigned to *fd*. The string 'Hi, folks!', which contains 10 bytes, is written to 'foo'. The file pointer is positioned at the beginning of file 'foo' by the **seek:** method with 0 as the file position argument. Note that 'foo' is always referred to by its file descriptor, not its name, when making system calls. Next, an instance of **String** is created and assigned to *answer*. The method **read:into:** fills the string with data from 'foo'. Try changing the size of *answer* to 8 and to 12 to see the different results in your System Transcript. The file is closed using the **closeFile:** method inherited from **AimSystemCall**. Closing frees the file descriptor, but 'foo' will still exist on the disk. To remove the file from the disk, you have to make another system call, like this:

```
OS remove: 'foo'.
```

Related Classes

AbstractSystemCall
AimSystemCall
Subtask

```
ExternalBinaryData variableByteSubclass: #Utsname

    instanceVariableNames:      ''
    classVariableNames:         'MachineDataIndex NameLength
                                NodenameDataIndex ReleaseDataIndex
                                SysnameDataIndex VersionDataIndex '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

Utsname provides accessing protocol for the following C structure.

```
struct utsname {
    char    sysname[9];
    char    nodename[9];
    char    release[9];
    char    version[9];
    char    machine[9];
}
```

The structure is documented under *uname(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

MachineDataIndex

NodenameDataIndex

ReleaseDataIndex

SysnameDataIndex

VersionDataIndex

Each C structure class variable holds the offset of a single field in the structure. The name of a class variable is constructed from a field name with the string 'DataIndex' appended. For example, the class variable **MachineDataIndex** holds the offset of the "machine" field.

NameLength <Integer>
Holds the constant, 9, of the char fields of the utsname structure.

Instance Methods

accessing

machine
Return the value of the structure field named machine.

nodename
Return the value of the structure field named nodename.

release
Return the value of the structure field named release.

sysname
Return the value of the structure field named sysname.

version
Return the value of the structure field named version.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the structure.

instance creation

uname
Return an initialized instance.

Rationale

The structure is used in support of the following UTek system call:

uname(2)

Related Classes

UTekSystemCall implements the system call listed above.

```
Object subclass: #VirtualStrikeFont

    instanceVariableNames:     'name ascent descent '
    classVariableNames:        ''
    poolDictionaries:          ''
    category:                  'Graphics-Support'
```

Summary

A **VirtualStrikeFont** represents a **StrikeFont** that has not been constructed in the image. A **VirtualStrikeFont** contains the name of the font that either **StrikeFont** knows how to read or can be synthesized. **VirtualStrikeFonts** are converted to their corresponding **StrikeFont** whenever they are referenced in a **TextStyle**, via the accessing method **fontAt:**.

Instance Variables

ascent <Integer>
The largest distance from the baseline to the top of any character in this font.

descent <Integer>
The largest distance from the baseline to the bottom of any character in this font.

name <String>
The font name.

Instance Methods

accessing

ascent
Return the instance variable ascent.

ascent: anInteger
Set the ascent to anInteger.

descent
Return the instance variable descent.

descent: anInteger
Set the descent to anInteger.

name
Answer the VirtualStrikeFont's name.

name: aVirtualStrikeFontName
Set the VirtualStrikeFont's name to aVirtualStrikeFontName, a String.

testing

isVirtual
Since the receiver is a VirtualStrikeFont, return true.

Class Methods

instance creation

name: aString
Return an instance with the name, aString.

Rationale

An instance of **VirtualStrikeFont** has been registered with the **FontManager**, but has not been loaded in the image. This class enables you to create a **TextStyle** with an array of fonts, but save Smalltalk memory by not actually having the fonts in your image until they are accessed for use. When a "virtual" font is converted to a "real" **StrikeFont** your image uses more memory.

Discussion

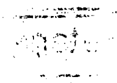
When an instance of this class is created, its name is assigned. Instances are created by the **TextStyleManager** *text style instance creation* methods, and that is the recommended way to create **VirtualStrikeFonts**. The *instance creation* method, **name:**, is called by a method in **StrikeFontManager**.

Accessing methods return or set the values of instance variables.

The *testing* method **isVirtual** returns true.

Related Classes

StrikeFont
StrikeFontManager
TextStyle
TextStyleManager



```
ExternalBinaryData variableByteSubclass: #Wait

    instanceVariableNames:      ''
    classVariableNames:         'CoredumpBitPosition CoredumpDataIndex
                                RetcodeDataIndex StatusDataIndex
                                StopsigDataIndex StopvalDataIndex
                                TermsigDataIndex TermsigMask '
    poolDictionaries:           ''
    category:                   'OS-Parameters'
```

Summary

Wait provides accessing protocol for the following C union.

```
union wait {
    int     w_status;    /* used in syscall */
    /*
     * Terminated process status.
     */
    struct {
    unsigned short   :16;              /* pad to make full 32 bits */
    unsigned short   w_Retcode:8;      /* exit code if w_termsig==0 */
    unsigned short   w_Coredump:1;     /* core dump indicator */
    unsigned short   w_Termsig:7;      /* termination signal */
    } w_T;
    /*
     * Stopped process status. Returned
     * only for traced children unless requested
     * with the WUNTRACED option bit.
     */
    struct {
    unsigned short   :16;              /* pad to make full 32 bits */
    unsigned short   w_Stopsig:8;      /* signal that stopped us */
    unsigned short   w_Stopval:8;      /* == W_STOPPED if stopped */
    }w_S;
}
```

The *wait* system call is documented under *wait(2)* in the manual *UTek Command Reference, Volume 2.*

Class Variables

CoredumpDataIndex

RetcodeDataIndex

StatusDataIndex

StopsigDataIndex

StopvalDataIndex

TermsigDataIndex

Each external data binary subclass class variable holds the offset of a single field in the structure or union. The name of a class variable is constructed from a field name, stripped of its prefix, with the string 'DataIndex' appended. For example, the class variable **CoredumpDataIndex** holds the offset of the "w_Coredump" field.

CoredumpBitPosition <Integer>
Holds the constant, 8, for the bit position of the core dump indicator bit in the w_Coredump field.

TermsigMask <Integer>
Holds the constant, 127, used to mask the termination signal bits in the w_Coredump field.

Instance Methods

accessing

coredump
Return the value of the union field named coredump.

retcode
Return the value of the union field named retcode.

status
Return the value of the union field named status.

stopsig
Return the value of the union field named stopsig.

stopval
Return the value of the union field named stopval.

termsig
Return the value of the union field named termsig.

printing

printOn: aStream
Print the receiver on aStream.

Class Methods

class initialization

initialize
Assign offset values to the class variables and define the size of the union.

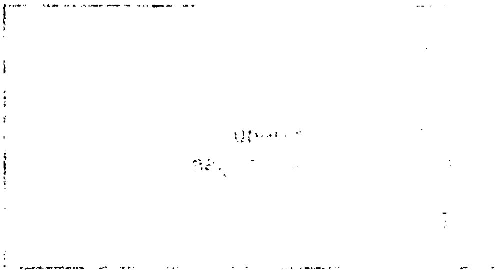
Rationale

The wait union is used in support of the following UTek system calls:

wait(2)
wait3(2)

Related Classes

UTekSystemCall implements the system calls listed above.



```
StringHolderController subclass: #WorkspaceController

    instanceVariableNames:      ''
    classVariableNames:         'WorkspaceYellowButtonMenu
                                WorkspaceYellowButtonMessages '
    poolDictionaries:           ''
    category:                   'Interface-Text'
```

Summary

WorkspaceController provides additional control for workspaces. Results of expressions in workspaces can be inspected.

Class Variables

WorkspaceYellowButtonMenu <PopUpMenu>
Yellow button menu in workspaces.

WorkspaceYellowButtonMessages <Array>
An array of symbols, each one being a selector that implements the corresponding item in WorkspaceYellowButtonMenu.

Instance Methods

menu messages

accept
Accept the workspace contents.

inspectIt
Treat the current text selection as an expression; evaluate it. Open an Inspector on the result.

Class Methods

class initialization

initialize
Initialize the yellow button pop-up menu and corresponding messages.

Rationale

This class allows the addition of middle-button menu items for workspaces.

Discussion

Using **WorkspaceController**, it is possible for you to add middle-button menu choices for workspaces. This implementation adds the "inspect it" menu choice. Expressions in ordinary workspaces and in the System Workspace can be selected and an Inspector will be opened on their results. Another menu choice you might add to workspaces is "file out" — it is available as a fileIn. Read about fileIns in the *Tektronix Smalltalk Users* manual.

Menu messages protocol includes the methods for middle-button menu choices. **Accept** has been reimplemented to save the contents of a workspace as text — this allows fonts used in the workspace to be saved with the text.

Class initialization protocol initializes the two class variables — the pop-up menu and corresponding messages.

Related Classes

Workspace
WorkspaceView