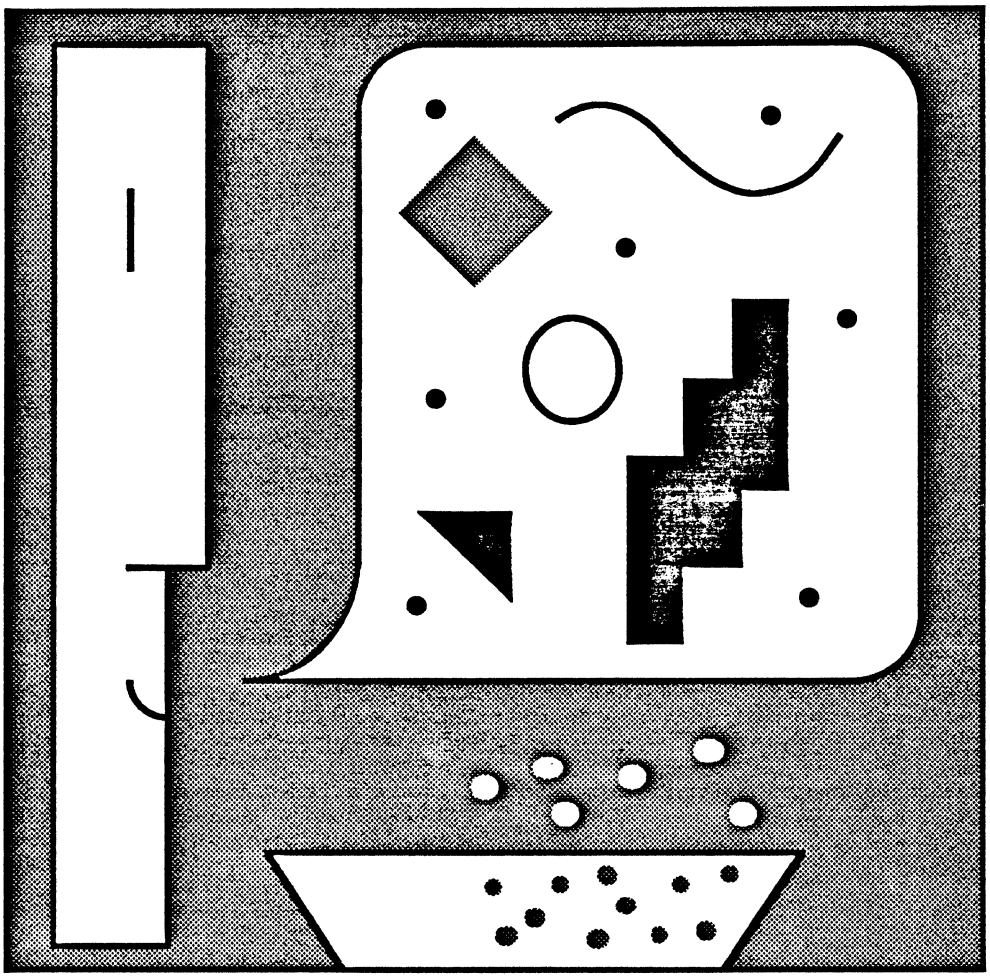

Smalltalk-80™ for the Macintosh™

The Smalltalk-80™ Programming System for the Macintosh™

August 1, 1985

About this Release

In response to requests from several universities, we have put together this "pre-product" release (0.2) of the Smalltalk-80 programming system which runs on the Macintosh XL and Macintosh 512K computers. We hope that this will enable more hands-on experience with Smalltalk in the universities, and that it will give other interested parties a chance to experiment with the system.

Release 0.2 supplants an earlier release (March 1985) that ran only on the Macintosh XL. Two images are supplied with the new release: level 0 and level 1. Both images incorporate the following improvements:

- Improved volume management capable of supporting file server access
- Automatic spelling correction for variable and message names
- Ability to handle multiple images on the same disk
- A single interpreter works on all machines and expands to use extra memory

Level 1 requires 1 MB or more of memory, and incorporates further improvements:

- Total object capacity has been raised from 24K to 32K
- Can expand to take advantage of 2MB memory on the XL
  (available from AST Research, Inc.)
- Able to run on some Macintosh computers with 1MB or more of memory
  (available from various manufacturers, but must be contiguous
  and must follow ToolBox memory organization)

Level 0 is a stripped-down image which will run on the Macintosh 512K. It is missing many of the classes in level 1, but it retains full support for browsing, editing, compiling and debugging. Although available memory is low (3K objects, 44K bytes free), it is adequate to support many introductory examples. If the Macintosh is connected to a hard disk, either local or shared over AppleTalk, users may access the full Smalltalk-80 source code. The system will also run with only the internal disk drive, using decompilation to view system methods; full source is retained for user-defined methods.

We have worked hard to make this implementation of the Smalltalk-80 system usable, and we feel that it can serve well as an introduction to object-oriented programming in general, and to the Smalltalk-80 system as well. Level 1 can even serve as a useful environment in which to build prototypes of simple systems.

Documentation and Support

We have included only enough documentation to get you started. If you are not already familiar with the Smalltalk-80 system, you will need further documentation not provided by Apple, such as the two Smalltalk-80 books by Goldberg and Robson (see references). Courses and training materials that relate to Smalltalk-80 are available from PPI, an organization that offers general training in object-oriented programming as well as specific courses on Smalltalk-80. For further information, contact:

Productivity Products International
27 Glen Road
Sandy Hook, CT 06482
(203) 426-1875

Apple cannot provide support for this pre-product release, since we are charging only enough to cover our costs of duplication and handling. If you should discover bugs in the documentation or in the system itself, your only recourse will be to work around them yourself. If a later version is released, there will be neither an upgrade price nor a guaranteed code migration path.

An Apple Smalltalk Newsletter

We want to know what you find to be good or bad about the system as released. We are especially interested in specific improvements or extensions which you find amusing or useful, or in any training materials which you may develop. Assuming there is sufficient response, we shall assemble such contributions in a newsletter for those who have purchased this release.

Since we do not have time to maintain an ongoing newsletter, we hereby solicit offers to take over publication of the newsletter following our first free issue. Assuming we find a volunteer to take over the newsletter, we will make the new arrangement known in the pilot issue.

Please send your comments and contributions to the address below:

The Smalltalk Group, Mail Stop 22-Y
Apple Computer
50525 Mariani Drive
Cupertino, CA 95014

The Future of Smalltalk at Apple

You will notice that this release violates Macintosh standards of user interface design in many ways. Windows do not repaint automatically, and mouse clicks may be lost. These are characteristics of the Smalltalk-80 system as originally licensed, and we have not tried to fix them for this early release.

One of the reasons why we will not support this release is that we are contemplating substantial changes to the system with the goal of integrating Smalltalk with the Macintosh ToolBox. The result will be a Smalltalk system that is smaller, faster, and more compatible with other Macintosh software. Compare the effect of executing

```
Pen new mandala: 30 diameter: 360
```

which draws lines without the aid of the ToolBox, with that of

```
QDPen new mandala: 30 diameter: 360
```

and you will see why we are excited about the combination of Smalltalk's flexibility with the speed of the Macintosh Toolbox.

In spite of the changes which we anticipate, we plan to stick with the language syntax, semantics, fileOut formats, and general programming interface embodied in this release of the Smalltalk-80 system.

Installing Level 1 Smalltalk with a Hard Disk

To run Smalltalk-80 on a Macintosh XL with MacWorks or on a Macintosh with a hard disk, your hard disk must be capable of holding the following files plus some 1300K of working space (roughly 3500K total):

1300K	Smalltalk-80.sources
577K	Level1.image
24K	Level1.changes
56K	Smalltalk (interpreter)
67K	12 examples (.st)
36K	DivJoin.

This release is shipped on seven Macintosh diskettes. All files needed to run on the Macintosh XL are on the first 6 diskettes. Because of the size of the first two files, they have been split up into six files named

1.Level1.image and **2.Level1.image**, and
1.Smalltalk-80.Sources through **4.Smalltalk-80.Sources**.

First move all the above files to your MacWorks volume, and then use the **DivJoin** program to reconstitute the two large files from their constituents. DivJoin is a program that allows very large files to be stored on diskettes by breaking them into pieces (each of which fits on a diskette) and reassembling them again.

To reconstitute the sources file, first double click on DivJoin. Once inside DivJoin, select **Open** from the **File** menu and choose **1.Smalltalk-80.sources**. Now select **Join this File** from the Div/Join menu to join all parts of the image together. If you need more space at this point, you can leave the DivJoin program and delete **1.Smalltalk-80.Sources** through **4.Smalltalk-80.Sources**. Again in the DivJoin program, open **1.Level1.image** and select **Join this File**. You have now reconstituted your files and can delete **1.Level1.image** and **2.Level1.image**.

You can minimize the amount of space needed during installation by moving and reconstituting Smalltalk-80.sources before moving any of the other files to your MacWorks volume. In this way you could get by with around 2500K of disk space. However, you will probably want more space for the convenience of keeping a backup copy of the image and changes, so as to avoid the full installation procedure the next time you want to start from a clean copy of Smalltalk.

Installing Level 0 on a Hard Disk

When using a 512K Macintosh with a hard disk, or in case you have a Macintosh XL with only 512K of memory, you will want to install level 0 on your hard disk. The procedure for this is just the same as for level 1, except that you want Level0.image and Level0.changes, instead of Level1. These files may are located on the last two diskettes in the set and, since Level0.image fits on one disk, there is no need to use DivJoin.

Installing Level 0 without a Hard Disk

If you do not have a hard disk, then Smalltalk must be run from diskettes. In this case there is no need for installation, since we have configured the last two diskettes as ready-to-run Smalltalk disks. We recommend that you first copy these disks and retain them as backups before proceding further. Having made a backup copy, you may delete DivJoin from disk #6 to make more space available (DivJoin is only used for hard disk installation).

Starting Smalltalk with a Hard Disk

To start Smalltalk after it has been installed on your hard disk, double-click or open the image icon (Level1.image or Level0.image). It takes about half a minute to load the image into memory, after which the Smalltalk screen is displayed.

Starting Smalltalk without a Hard Disk

First boot your Macintosh from disk 6 (system, interpreter and Level0.changes). If you have an external disk drive, place disk 7 (with Level0.image on it) in the external disk drive. If you only have the internal disk drive, eject disk 6 and insert disk 7. To start Smalltalk double-click or open the icon for Level0.image. With only one disk drive, you will be asked to change disks a few times while the Smalltalk interpreter loads the image. It takes about a minute to load the image into memory, after which the Smalltalk screen is displayed.

Once in Smalltalk

Once in Smalltalk, saving your state and quitting are done by selecting options from the "screen menu." Point in a gray area where there is no window and hold the mouse button down (don't let go!). The screen menu will appear, presenting a choice of several global operations, such as restoring the display, opening new windows, making a snapshot, and quitting. Move the cursor to the command you want -- or outside the menu if you don't want any of them -- and then release the button.

Making a **snapshot** is like the **save** operation in most Mac software, but it causes the entire state of your Smalltalk -- not just the active window -- to be written back onto the image file, after which you may continue working in Smalltalk. If you make an unrecoverable error, or if Smalltalk should crash, restarting Smalltalk will return you to the state at your last snapshot or quit-with-save.

The **quit** option is like the quit operation in most Mac software. It first prompts you to choose whether to save the changes you have made. If you choose **no**, then the next time you run Smalltalk, none of the changes since you started (or since your most recent snapshot) will remain. If you choose **yes**, then the next time you run Smalltalk everything will be just as you left it - all the windows will be where they were, even interrupted processes will remain in a state of suspended animation. [Note, therefore, that if you suspect your Smalltalk to be flawed, you should **not** snapshot or save changes at quit]. Open files will be closed by the snapshot and quit operations, and they will be re-opened automatically upon restarting Smalltalk.

System Workspace and 'Hello' File

Many useful executable expressions have been collected in a window called the System Workspace in the Level 1 image. Because this text consumes valuable space, it has been omitted from the Level 0 image, but we have included an annotated copy as an appendix to this documentation.

We have also included several interesting code fragments in a file named 'Hello'. Here is what you need to do to open a window on that file. Once in Smalltalk, click the mouse in the window in the upper right corner of the screen. If it is not already selected, select the item **(FileStream oldFileNamed: 'Hello') edit.**

Selection is done just as in MacWrite. If there is no such text, you can type it, just as with MacWrite. When the text has been selected, slowly move the cursor into the white scroll bar at the left of the text area. In the rightmost edge of this area, the cursor should take on the shape of a little menu. When that happens, click the mouse button and hold it down. A menu will pop up on the screen, from which you should choose 'DoIt', and then release the mouse button. This causes Smalltalk to compile the selected expression and execute it, which has the effect of creating a new window viewing the contents of the file named 'Hello'.

When the file has been opened, Smalltalk needs you to specify a rectangular area for the window on the screen. The cursor should change to a bracket shape, prompting you to click where you want the upper left corner of the window, then it should change to another bracket prompting you to click where you want the lower right corner of the window. Once you have done this, the text in the file will appear, and it should tell you everything else you need to know.

Save Frequently

The Smalltalk-80 system is by nature a very malleable programming environment. Some of the changes you make may leave you with a dead system. A good rule is "Don't play for more than you would be happy to lose." Even if you never make a silly mistake, the power might go off, or you might find a bug that we don't know about.

Differences from Other Smalltalk-80s

We have found it necessary or convenient to depart from the specification of Smalltalk-80 which is documented in the Smalltalk-80 literature. We list on this page the chief differences that will be noticed by users with experience on other systems.

Mouse Buttons

You can click in right edge of a scroll bar for the yellowButton menu.
The Option key works as a shift to give "yellow" button when otherwise needed.
You can click window title tabs for the blueButton menu.
The Enter key works as a shift to give "blue" button when otherwise needed.
Click in the gray area outside all windows for the screen menu

Keyboard and Editing Operations

Apple-X, C, V are keyboard synonyms for cut, copy, paste.
Use shift-6 to get up-arrow, shift-minus to get left-arrow.
Use Apple-period for keyboard interrrupt (used to be control-C).
Text selection can be extended by shift-clicking.
Typing does not interfere with the paste buffer.
Paste does not leave an extended selection.

The inspectors in this Smalltalk have an extra panel at the bottom. Here you can type and evaluate expressions in the context of the inspected object, without their being overwritten by the display of field values.

Spelling correction in this Smalltalk is automatic, rather than voluntary, and you have the opportunity of choosing among the closest matches. The correction algorithm favors messages that begin the same, but you can choose **try harder** to include a wider variety of messages in the search.

System Sources and File Servers

Smalltalk always expects the system sources to be named **Smalltalk-80.sources**. If there is no file by that name, Smalltalk will use decompilation to show source code for system methods. Each time Smalltalk starts up, it checks every mounted volume for the files it needs. Consequently, if your Macintosh is on a network, and if there is a disk server on that network, and if that disk server is "mounted" as a volume on your Macintosh, and if there is a copy of Smalltalk-80.sources on the disk server. . . guess what. . . Smalltalk will open the file over the network for its access to sources. In this way it is possible for many Macintoshes without hard disks to browse full source code if they are connected to a network with a disk server.

FileIn and FileOut

You can save and load parts of your Smalltalk system -- individual methods, classes, groups of classes, and so on -- in a text file format called "fileOut format". The fileOut and fileIn operations are available in browser menus and as messages of which there are examples in the System Workspace window. These files can be moved from one image to another, and can be read by MacWrite. You can even edit them in MacWrite if you are **very** careful not to disturb the file format conventions, and save as text-only.

Note that browsers do not properly reflect changes made in other browsers or in fileIns. It may be necessary to reselect an item or, in some cases, even to close and reopen the browser. The **update** option in the leftmost panel causes a browser to show new system categories which have been created by another browser or by a fileIn.

Smalltalk State

Two files, an image and a changes file (by convention named **xxx.image** and **xxx.changes**), <u>together,</u> constitute the full state of a given Smalltalk system.

Therefore, to completely back up the state of your work in Smalltalk, you must save copies of both of these files. You may rename the copies at will, but you must abide by the convention that the two names begin the same, and end with **.image** and **.changes**.

The Image and the Changes File

If you become a serious user of Smalltalk, you will find it useful to understand how the changes file (Smalltalk-80.changes) is used. Each time you **accept** a method, or execute an expression (**do it**) in Smalltalk, a copy of the source code in question is appended to the changes file. This file provides the necessary data to later browse the source code for the changes you have made.

Since the changes file constitutes a log of changes you have made, it can be very valuable in case of disasters. In the System Workspace you will find the executable statement, **Smalltalk recover: 5000**. This will copy the last 5000 characters of the changes file to a temporary file and prompt you to open a window on that text. In that window, you can select important lost information, and re-execute it to recreate the changes you lost. The changes file is in fileOut format. The file window includes a menu command **fileItIn**, which will cause a fileIn of the *currently selected* portion of the file. In the case of method definitions, the **methodsFor:** line which precedes the definition must be included, as well as the exclamation points that delimit the end of the method definition item. The longest recovery that can be effected in this way is currently the practical file size limit of 16000 characters imposed by the file edit window.

The changes file grows every time you run Smalltalk. If you redefine a method twenty times, there will be twenty copies of it in the changes file. Also, even when you quit Smalltalk without saving changes, the changes file will have grown as a result. Eventually, it will be necessary to condense the changes file - that is, to write a new copy which contains only the code currently accessible from the image. This can be accomplished by executing the statement **Smalltalk condenseChanges**. It is advisable to make a backup copy of your state (**.image** and **.changes**) before condensing the changes file. The process can take a long time (10 minutes or more), depending on how many changes you have made in your image.

Users without hard disks will have to pay particular attention to the amount of space remaining on the disk used for changes. The condensing process itself requires additional space for the condensed copy, so one should ideally condense the changes file whenever it becomes as large as the free space remaining on the disk. If one is attempting serious work without a hard disk, this restriction may become a problem. In such cases, you may fileIn the goodie named **dropChanges.st**, and then execute **Smalltalk dropChanges**. The result will be to reclaim all space in the changes file, but future browsing of the changes made up to that time will only present a decompiled version of the code - in other words temporary names and comments will be lost. If you must resort to this solution, you should probably save fileOuts of the code you care about, so as to retain a copy of the code with commments.

The Change Set

Besides the changes file itself, the Smalltalk-80 system provides another completely independent mechanism for recording what changes have been made. While the changes file is a linear log which records all the text involved, the changes set holds *references* to changes made to the classes and methods within the system. It is used in conjunction with the fileOutChanges method. For example, executing

(FileStream newFileNamed: 'Jan26/Changes.st') fileOutChanges

will cause every change recorded in the change set to be written out to the named file. Such "fileOut" files are a much more compact way of saving a Smalltalk project than saving the whole image. If you use project windows, a separate change set is maintained for each project window in your Smalltalk. Note that the statement

Smalltalk noChanges

can be used to reset the current change set. If three more changes are made thereafter, only those three changes, and no previous changes, will be recorded.

Whenever a fileout includes a class **initialize** message, it will also automatically generate a statement to invoke this method at the end of the file.

Memory Space

Space is very tight on 512K machines, and therefore Smalltalk may be uselessly cramped or may crash if other systems compete for its space in memory. In particular, we know it will crash if MacsBug is loaded, and it is very cramped if a ram-resident file system is used.

You may be interested in the question of how to make more free space available. We offer a few tools and a few suggestions, and we request that, if you have good success, you let us know what you did, so that we can pass it on to other users.

For measurement, you will find **Smalltalk printSpaceAnalysis** useful. It lists on a file roughly how much space (objects and words) is consumed by each class and by instances of that class.

Here are some suggestions about how to reclaim significant amounts of space:

Remove FileLists (simply execute FileList remove).

Remove the debugger step and send commands, and any other code which uses the simulation in class ContextPart. If you can't live without step and send, then implement breakpoints (a useful thing to do anyway), and then use these to restore the operation of step and send.

Make MessageSet, and maybe even the Debugger, be a subclass of Browser so that they can share code for printing, fileOut, senders and messages.

If you have a hard disk, remove the decompiler (simply execute Decompiler remove). One minor problem is that the decompiler is currently used for viewing the code of do-it methods from the debugger. If you can't live without this, then figure out how to log the text of do-it methods in the changes file and give them a proper source code pointer.

After removing significant parts of the system in this way, you can use **Smalltalk removeUnSentSelectors**. This method locates any messages which are implemented, but nowhere sent in the system, and then removes then. It is useful to run this method several times in a row (like 5), because each method removed may render other methods inaccessible. This method takes a long time (from 5 to 25 minutes) to run.

Finally, you should run **Symbol rehash**, which will reclaim all the Symbols which are no longer in use as a result of the removal of methods.

Memory Management

The interpreter furnished with this release uses garbage collection rather than reference counts to reclaim unused storage. For this reason, it is not necessary to break circular structures in this implementation, though you will find many places where such "release" code still exists in this image. You will notice occasional pauses in the operation of the system when garbage collection takes place. Some of these take longer than others, and we are working hard to make them all shorter.

A side-effect of the incremental garbage collection scheme used in this implementation is that **Behavior allInstances** and related enumeration messages will sometimes enumerate objects that are no longer truly accessible. If you want to be sure of the results, execute **Smalltalk garbageCollect** immediately prior to such enumeration.

Even with garbage collection, it is still possible to run out of memory. If there is enough space in your system, a NotifyWindow will appear telling you that space is low. The stack of senders will allow you to check if the cause was endless recursion. It is generally advisable to close the NotifyWindow, rather than proceeding, since there will be no soft error recovery possible after proceeding. If the problem was not an endless recursion, you should consider how you might have unintentionally consumed a lot of storage. In some cases you may have to jettison some other competing objects in the system. Level 1, contains a method **SystemDictionary deleteClasses** which performs such a jettison operation.

You can find out how much memory is available in your system by printing

Smalltalk spaceLeft

which returns the number of free objects and number of free words of data. The minimum safe operating margin is around 1000 objects and 4000 words of data.

The Macintosh File System

This implementation of Smalltalk-80 includes an interface to the Macintosh file system. Instances of MacVolume refer to the various disk directories. Files can be opened by sending messages to FileStream, such as **FileStream fileNamed: 'Hello'**. Several examples appear in the System Workspace. There is a default volume on which a file name will be sought, but you can override this by including a prefix (with colon) in the file name, as in **'Internal:fileName'**. If you don't know what volumes are available, you can use ? in place of the volume name, as in **'?:Hello'**, and

Smalltalk will present you with a menu of the currently mounted volumes and their aliases. You can eject a volume by sending it the message **eject**, or by choosing **eject disk** from the screen menu. If there are files open on this volume, you will be prompted to close them before ejection.

Files in the Macintosh file system are modelled by instances of MacFileStream. These respond to normal stream protocol, a well as to several other messages which can be found by browsing through the implementation. If you overwrite an existing file, it is necessary to send the message **shorten** to the fileStream prior to closing, lest part of the old data remain after the end of the new contents.

The default type of files created by Smalltalk is **TEXT**. This means that they can be read by MacWrite as text-only. If you wish to give other properties to a file, you can use the method **setType:creator:** to do so, following the general approach used in **MacFileStream typeTEXT**. If you wish to write formatted text that can be read by MacWrite, you will have to implement the MacWrite document format which is fairly complex. Please let us know if you do it.

Further information about the interface to the Macintosh file system appears in the appendix *Using the Volume Oriented File Package.*

The Macintosh ToolBox

This version of Smalltalk includes an interface to the Macintosh ToolBox routines. You can call any Macintosh ToolBox routine as long as the routine has a trap number (some do not and so must be reimplemented in Smalltalk, which we have not done yet) and its arguments are simple enough. The message is constructed as follows:

the receiver is **Mac**, the sole instance of class Macintosh,
the first keyword is the name of the routine
followed by the first argument;
each additional keyword is the formal name of its argument
followed by the argument itself.

Examples are:

```
Mac penNormal.
Mac textSize: 14.
Mac offsetRect: someRect dx: 22 dy: -33.
```

A nearly complete exercise of the QuickDraw routines can be found in

ToolBox-Support>Macintosh>quickdraw sample>drawStuff

and the routines which it calls. A complete description of the ToolBox interface can be found in the appendix *Smalltalk ToolBox Access.*

About this Version of Smalltalk-80

This version of the Smalltalk-80 system, known as *version 1,* was licensed from the Xerox Corporation as part of an early collaboration on Smalltalk development. Since that time, Xerox has made changes and additions to their version which they now offer for general licensing as *version 2.* While there are several differences between these

two versions, the language is nearly identical, and most of the kernel programming tools operate similarly. The principal features of version 2 that are not supported in this release include:

Multiple Inheritance support (not actually used in version 2)
Sticky browser selections
Changes file browser
"Pluggable" viewing protocol

Some of our customers already have a license for the Xerox version 2 image. It would be nice if this image could be adapted to run on the Apple interpreter so that there would be complete compatibility between Smalltalks on different machines within such organizations. If you are such a customer, you should at least let us know that you want such an image. More importantly, if you would like to make it happen, write us a letter to that effect. We are prepared to furnish the information you would need to produce such an image. This could make an interesting student project, and the result would be a valuable contribution to the Smalltalk-80 community. The current Xerox licensing policy provides for distribution of such a modified image among others who have purchased a Xerox license.

Sample Smalltalk Files

Included with your Smalltalk system are several Smalltalk files ("goodies") which add interesting or useful capability to your Smalltalk. Execute **(FileStream oldFileNamed: 'filename.st') fileIn** to read any of the sample files into your system. The file names are not case-sensitive, but spaces are significant. The **.st** naming is only a convention - it makes it easy to browse all Smalltalk files using ***.st** in the top pane of a file list.

Printing.st includes a definition for **MacPrintStream**. This supports access to an Apple Imagewriter connected to the printer port. It also adds a menu option for printing most windows. Printing from the window title menu produces a bitmap of the window in question. Printing from a text pane menu produces text with an attempt to support the unusual Smalltalk characters. The class method **examples** includes an expression for printing the bits of an arbitrary screen rectangle. Some such bitmaps will be too wide for the printer; use a wide-carriage printer, or figure out how to print landscape bitmaps (actually easier than the portrait format supported by MacPrintStream). Imbedded in this file is a method for rotating bitmaps by 90 degrees, which might be otherwise useful.

FFT.st defines a class which will perform a one-dimensional complex Fast Fourier Transform of data held in Smalltalk arrays. The example method **test** illustrates the use of this capability. This code computes a complex Fourier transform; what many people expect is a "power spectrum" which can be derived by summing the squares of corresponding real and imaginary components.

Fractal.st contains code for producing three-dimensional surfaces based on fractal geometry. Execute **Fractal example** for an example.

Toothpaste.st alows one to draw shaded worm-like curves on the screen using a "brush" that looks like a highlighted sphere. Execute **Form toothpaste: 30** and then paint with the cursor. Use option-click to stop.

Web.st is another cute drawing program. It uses a model which lags behind the current mouse position, and then draws lines between the lagging cursor and the current cursor. Execute **QDPen new web** and then draw with the mouse. Click to erase the screen, and option-click to stop.

Macintosh-quickdraw.st is a full set of QuickDraw call definitions. This makes all the QuickDraw messages visible, which they are not with the current space-saving generic lookup mechanism (see appendix). At the cost of some memory space, it thus makes QuickDraw messages easier to browse.

DictionaryInspectors.2.st allows direct access to keys and values when inspecting dictionaries. This goodie has already been included in the Level 1 image.

MacPaint.st defines a method which will create a graphics file readable by MacPaint. The comment at the end of method Form macPaintOn: gives a sample invocation. The method is already included in Level 1, but its example comment is in error. If you fileIn this goodie, it will correct the comment.

RS232.st defines a few methods which support input and output via the two Macintosh serial ports. One of these includes a large comment explaining how to set baud rate etc., and another is a sample application which downloads text (eg from a lap computer) to a Mac file.

Retrieve.st allows access to the full system sources, even if your Macintosh does not have a hard disk. If you have browsed to a given method (decompiled), and wish to see the full system source (as you would with a hard disk) choose **retrieve** (a new option) from the code pane menu. You will be asked to insert one of the disks with the divided source file on it, and then the full code will appear. This technique will fail if you happen to choose one of the three methods that straddle breaks between the file divisions.

DropChanges.st is for the fearless programmer whose **.changes** file has grown too large (even after condensing), yet who still wants to continue making changes. One could, of course, simply delete the changes file, but this will prevent the further recording of changes. Instead, by executing **Smalltalk dropChanges**, the current source code will be discarded (save it first, if you care), but the change file will continue to record further development.

Known Bugs

If you collapse the Transcript, or have multiple views open on it (we haven't tracked down the details), Transcript operations occasionally fail or get hung in a loop. In case of such a failure, it should suffice to interrupt with Apple-period, then execute **Transcript clear** in some other window.

The Smalltalk simulator, which is used for **step** and **send** in the debugger, is incapable of single-stepping through process-switching operations such as **Processor yield**. Try to be aware of this situation, as it will crash in a way that can be cured only by rebooting.

References

The four principal references currently available for the Smalltalk-80 system are:

BYTE Magazine, August 1981. This issue is a Smalltalk-80 special, and includes articles on the language, graphics and other aspects of the system.

Smalltalk-80, The Language and its Implementation by A. Goldberg and D.Robson, Addison-Wesley 1983.

Smalltalk-80, The Interactive Programming Environment by A. Goldberg, Addison-Wesley 1984. Contains a summary of the language as well.

Smalltalk-80, Bits of History, Words of Advice, edited by Glenn Krasner, Addison-Wesley 1983. Specific to implementation techniques - not useful as a tutorial.

Appendix 1
Using the Volume Oriented File Package

The Volume Oriented File Package that is in the latest release of Smalltalk (June 18th image and beyond) is slightly different than the previous file package that Smalltalk used. Highlights are multiple volume support, source and changes file hunting, user aliases, and user defaults.

The Basics

Disk and **Diskette** no longer exist. The most common operations on files can be translated to the new system as follows:

Old File Package	New File Package
Disk file: 'foo'	FileStream fileNamed: 'foo'
Disk oldFile: 'foo'	FileStream oldFileNamed: 'foo'
Disk newFile: 'foo'	FileStream newFileNamed: 'foo'
Disk filesMatching: '*.st'	FileDirectory filesMatching: '*.st'

All file names are of the format: **<volume prefix>:<name>**. The volume prefix and colon may be left off to access the default volume (see below for more on the default volume). Valid volume prefixes are **Internal**, **External**, **MacDefault**, and the names of whatever volumes are mounted. In addition, **?** may be used as a volume prefix to get a menu of all volume prefixes to choose from.

New volume prefixes may be aliased to existing volumes via the message **alias:to:** sent to **FileDirectory**. For instance **FileDirectory alias: 'Diskette' to: 'Internal'** will set up **Diskette** to be a valid volume prefix referring to the same volume as prefix **Internal** does. Note that there is no difference between an alias and the original volume prefix however, aliasing is not transitive (i.e. if you alias a to b, then b to c, a is now different from b and c). Volume prefixes may be removed with the message **unalias:** sent to **FileDirectory**.

The volume prefixes **Internal**, **External**, and **MacDefault**, are initially set up to be the internal diskette, the external diskette, and the Macintosh default volume (where the Smalltalk application was started from) respectively. These volumes are known as the fixed volumes and have some special properties. You of course may alias other prefixes to these volumes, and may alias the names to other volumes, however, the properties of being fixed volumes stays with the volumes, not the prefixes.

Any volume may be made to be the Smalltalk default volume, which is used when no volume prefix is specified. This can be done by sending **default:** to **FileDirectory**. For instance **FileDirectory default: 'External'** sets the default to be the external drive. The default is just like an alias, and so if you re-alias the volume prefix, the default will not have changed. Do not confuse the Smalltalk default with the Macintosh default volume (which **MacDefault** is initially a volume prefix for), they have no connection to each other. However, one can make the Smalltalk default the same as the Macintosh default with the expression: **FileDirectory default: 'MacDefault'** which is how the system is initially set up. Finally, you can have no default by sending

the message **noDefault** to **FileDirectory** which cause the system to ask you to choose a volume every time you do not include a volume prefix in a file name.

When the system starts up it aliases all mounted volumes using their names as volume prefixes (so you can use the name of a volume directly). As you execute if new volumes become known (via inserting a diskette for example) then they will be aliased in the same way. To eject a disk type **command-e** and then select the volume you want to eject from the pop up menu that appears. Ejecting now merely places the volume off-line (it used to unmount it) and so files may be still open on an ejected disk. Upon shutdown, all volumes are flushed and all but the fixed volumes are forgotten. In the process, all files that were open on forgotten volumes are closed and placed in a state such that further activity on them will fail. Files on fixed volumes will be re-opened when the system comes up (if still referenced) *but no check will be made to see if it is really the same volume.*

The Frills

Smalltalk will now hunt for the sources and changes files when it starts up. The name of the changes file is taken to be the same as the image file with **.image** stripped off (if it's there) and **.changes** tacked on. Note that the system forms the name at startup and so you can rename your image from the **Finder** so long as you rename the changes file too. Furthermore, you can now have multiple images on you disks. If you double click on an image, then it will be opened by **Smalltalk** If you just double click on **Smalltalk** then it will try to open an image named **Smalltalk-80.image** and bomb if it isn't there.

The changes file and sources files have now been untied. Now you can have one without the other. Hence, never set **SourceFiles** to **nil** if you want to run without sources (occasionally useful), do **SourceFiles ~ Array new: 2** instead. Furthermore, unlike before, the system will try to find the sources and changes each time you startup (before, once you ran without them, you had to manually reinstate them).

The Guts

The following messages can be sent to class **FileDirectory**. They cover all of the old functions that were performed by **Disk** and then some.

aliasesDo:	enumerate all the associations between volume prefixes and actual volumes
checkName:fixErrors:	check the first argument as a legal file name if it is answer it, if isn't and the second argument is true answer a fixed-up version of the name, else error
convertName:to:	convert the first argument to a volume

	and a file name and then evaluates the second argument with them as the two values
do:	enumerate all volumes, will duplicate some, includes the default
filterWith:	like **do:** but replaces each volume with the result of the block
fromUser	puts up a menu and lets the user select a volume prefix, answers the volume
isLegalFileName:	answers **true** or **false** based on whether or not its argument is a valid file name

Much of the pre-existing generic file package was removed (classes **File** and **FilePage** for starters) as it was actually quite Xerox specific. As a consequence, adding a another file package (in addition to Macintosh files) may be quite difficult if it is lower level than the Macintosh file model. However, since I don't see us using another file system with our Smalltalk I don't think that this is a big worry (famous last words...)

Most of the standard Macintosh volume operations have been defined for instances of **MacVolume**. In particular the messages **eject**, **flush**, **mount**, and **unmount** exist. Furthermore **close** and **open** exist for compatibility with the abstract Smalltalk model; they perform a **flush** and a **getVolInfo** respectively. Smalltalk tries to keep you from wreaking havoc on the Macintosh file system, but is easy to get around. For instance never unmount the Macintosh startup volume (go on, make my day...).

Appendix 2
Smalltalk ToolBox Interface Details

General Comments about Smalltalk Toolbox access:

Within class Macintosh in your standard Image there are several method categories of interest. Category *globals* contains methods to access various quickdraw globals. Category *quickdraw calls* contains the calls we thought should be implemented using primitive 160 to make them a little faster. The category *quickdraw sample* contains a sample program that does various things with quickdraw. This is a good sample program for you to see how to use the Toolbox using Smalltalk. To run it, execute **Mac drawStuff**. The *memory inspect* category contains a method, **d0Result**, that allows you to see what the D0 result was after a Toolbox call that returns result information in D0. There are also methods in *memory inspect* for looking at any particular memory location.

If you need to define some additional PascalRecords, look at the existing definitions within class categoryToolbox-Support to see how to deal with things like nested Pascal RECORDS.

There are 8 types of data currently being passed between SmallTalk and the Toolbox. They are Integer, LongInt, Boolean, Point, Rectangle, String, Pointer and Handle. Each of these types is stored in Smalltalk format while using it within Smalltalk code, then it is converted to Pascal format within a call to the Toolbox. Pointers and Handles are stored within a Smalltalk class called **PascalRecord**. The PascalRecord class was created specifically to deal with Pascal RECORDs and other pieces of Toolbox data that are accessed by either a pointer or a handle. More details on PascalRecords later. Within the Toolbox interpreter there are 16 low level routines to deal with conversion (both ways) of these 8 data types. These routines are called from Primitives 160, 161 and 162 through groups of intermediate level routines that set things up for the low level conversion routines to handle.

Primitive 160 is used by defining a method within Smalltalk for each Toolbox procedure you want to call. This method contains parameter information in its literal frame. There are a few methods defined in this way in the *quickdraw calls* category of class Macintosh in your image. If you haven't defined a Toolbox call using primitive 160, then primitive 162 is called automatically through the **doesNotUnderstand** method of class Macintosh. It looks up the name of the method in a table of Toolbox calls where it then gets the correct parameter information for the call. The lookup uses only the first 8 characters of the name and ignores case. Once they obtain parameter information, both primitive 160 and primitive 162 call a interpreter routine 'CallTraps' to do the actual data conversion and perform the call. For Toolbox calls that you use very frequently, you may want to use primitive 160 since it is slightly faster. Most Toolbox calls will work fine already though using primitive 162 and that is what we recommend for probably 90% of the calls. Primitive 160 uses more space than primitive 162 so use it sparingly.

There may be some errors in the table used by primitive 162. If you have problems with any Toolbox call that uses primitive 162, implement the call yourself with primitive 160 then please let us know about the problem! You also need to know that currently Smalltalk can only access Toolbox calls that are made through traps. This includes all of Quickdraw and most of the Toolbox calls. You may find a few that won't work though. These will be available in a future release!

Primitive 161 (used to read from and store into the fields of Pascal records)

A PascalRecord in Smalltalk has three fields:

handle	<LargeInteger>
pointer	<LargeInteger>
bits	<ByteArray>.

In this memo, **PascalRecord** means one of these above Smalltalk objects but **Pascal RECORD** means a Pascal format piece of memory used to store Pascal format fields. This is usually associated with a Pascal RECORD TYPE definition. There should be a subclass of PascalRecord defined in SmallTalk for each type of Toolbox Pascal RECORD you need to access. For example: BitMap, GrafPort, Region, etc. A PascalRecord should be in only one of these four states:

1. handle = Nil, pointer = Nil and bits = Nil. (no data space is allocated for this record)
2. pointer = a SmallPositiveInteger or LargePositiveInteger, handle = Nil, bits = Nil. (data space for this record is in a fixed location within a Toolbox heap)
3. handle = a SmallInteger or LargePositiveInteger, pointer = Nil, bits = Nil. (data space for this record is in a relocatable location within a Toolbox heap)
4. bits = a Smalltalk ByteArray containing an even number of bytes, pointer = Nil, handle = Nil. (data space for this record is in a relocatable location within the Smalltalk heap)

From one of these states, the conversion routines produce the correct information for Pascal as follows:

Conversion to a Pascal pointer:

```
if handle ~= nil then follow it and pass the pointer
if pointer ~= nil, then pass the pointer
if bits ~= nil, pass the address of the bits within the ByteArray,
    offset to skip ST's length and class fields
otherwise return a primitive error
```

Conversion to a Pascal handle:

```
if handle ~= nil then pass it
if pointer ~= nil, then create a handle on the stack and pass it
it bits ~= nil, create a handle and pointer on the stack pointing the
    the address of the bits within the ByteArray and pass it
otherwise return a primitive error
```

Here is an example of how we use PascalRecords and primitive 161. If we want Smalltalk to be able to read and write the portRect field of grafports, we would define the following two methods in class GrafPort: (GrafPort being a subclass of PascalRecord)

```
portRect
    "Pascal =        rect := aPort.portRect;"
    <primitive: 161 recordOffset: 16 type: 'R'>
    ^self primitiveFailed
portRect: rect
    "Pascal =        aPort.portRect := rect;"
    <primitive: 161 recordOffset: 16 type: 'R'>
    ^self primitiveFailed
```

This tells primitive 161 that at offset 16 (bytes) within self (the Pascal RECORD pointed to by the PascalRecord parameter on the Smalltalk stack) there is a rectangle field. If there are two parameters (portRect: rect) the primitive knows it is a write. It then converts the SmallTalk format rectangle on the top of the Smalltalk stack into an 8 byte Pascal format rectangle and stores this at offset 16 within the Pascal RECORD. If there is only one parameter, primitive 161 converts the rectangle at offset 16 from a Pascal format rectangle into a Smalltalk format rectangle which it passes back on the Smalltalk stack.

Here is a table of the 8 types of data that can be passed in the **Type Descriptor** field to primitives 161 and 160. In the case of primitive 161, there will only be one parameter described in the **Type Descriptor** field. As you will see below, primitive 160 has one type for each parameter in a particular Toolbox call.

type code	**Toolbox Type**	**ST class** (data formats are different)
I	Integer	SmallInteger, LargeInteger up to 16 bits
L	LongInt	SmallInteger, LargeInteger up to 32 bits
B	Boolean	Boolean
P	Point	Point
R	Rectangle	Rectangle
S	String	String
D	Pointer	PascalRecord (think d for direct pointer)
H	Handle	PascalRecord

Primitive 160 (used to call Toolbox register or stack based routines through the Toolbox Trap mechanism)

Here is some sample code to create a grafport, use it to frame a rectangle, and release the port. Mac is a Smalltalk object of class Macintosh which implements Macintosh Toolbox methods by calling primitive 160 with the correct Trap number and Parameter types as arguments. The Smalltalk code:

```
| port oldPort |
oldPort <- Mac getPort.
port <- GrafPort newM.          "newM for data on the Mac heap, newS for data on the
                                Smalltalk heap"
Mac openPort: port.
Mac setPort: port.
Mac frameRect: (10@20 rect: 100@100).
Mac setPort: oldPort.
Mac closePort: port.
```

You do not really need to understand how to set up calls to primitive 160 since primitive 162 already has a table containing parameter information for all the Toolbox traps. This documentation is only provided in case you want to speed up a few of the calls you use over and over again within a loop. Here are some of the methods in class Macintosh which get called in the above example:

```
getPort
    "Pascal =        PROCEDURE GetPort (VAR gp: GrafPtr);"
    <primitive: 160 trapA: 16rC74 type: '--VD'>
                    "trap number"   "no function, ignore receiver, VAR pointer"
    ^self primitiveFailed
```

```
openPort: port
    "Pascal =        PROCEDURE OpenPort (gp: GrafPtr);"
    <primitive: 160 trapA: 16rC6F type: '--D'>
                    "trap number"   "no function, ignore receiver,   pointer"
    ^self primitiveFailed
```

```
frameRect: rect
    "Pascal =        PROCEDURE FrameRect (r: Rect);"
    <primitive: 160 trapA: 16rCA1 type: '--R'>
                    "trap number"   "no function, ignore receiver,   rectangle"
    ^self primitiveFailed
```

```
closePort: port
    "Pascal =        PROCEDURE ClosePort (gp: GrafPtr);"
    <primitive: 160 trapA: 16rC7D type: '--D'>
                    "trap number"   "no function, ignore receiver,   pointer"
    ^self primitiveFailed
```

Type Descriptor (**Type**) Details:

For primitive 160, the **Type Descriptor** parameter describes the same **type codes** as listed in the table within the primitive 161 description above.

When a **type code** character represents a VAR parameter, precede it by the character 'V'.

When a parameter is passed in a register to a Toolbox routine instead of on the stack, the Type Descriptor string has to let primitive 160 know about this. The Toolbox uses three registers (D0, A1 and A0) to pass and return parameters. In the Type Descriptor string, the **register codes** are:

D0 = '0'
A1 = '1'
A0 = '2'

The register code character optionally follows the normal **type code** character.

Unlike primitive 161 which always expects a **Type Descriptor** string of length 1, primitive 160 expects a **Type Descriptor** string of the following format:

<Type Descriptor> = <function result type><receiver type><parameter info>

<result type> =	**type code** or '-' if not a function. The code character can optionally be followed by a **register code** if the function result is passed back from the Toolbox in a register.
<receiver type> =	**type code** of receiver if receiver is used as a parameter or '-' if receiver is ignored.
<parameter info> =	a **type code** character for each parameter other than the receiver. Each type code character is preceded by a 'V' if that parameter is a VAR parameter. Each type code character can optionally be followed by a **register code** if the parameter is passed to the Toolbox in a register. Nothing if there are no parameters.

More examples:

```
globalToLocal: pt
    "Pascal =        PROCEDURE GlobalToLocal (VAR pt: Point);"
    <primitive: 160 trapA: 16rC71 type: '--VP'>
                "trap number"  "no function, ignore receiver,  VAR point"
    ^self primitiveFailed
```

```
sectRect: srcRectA srcRectB: srcRectB dstRect: dstRect
    "Pascal =        FUNCTION SectRect (srcRectA, srcRectB: Rect;
                                        VAR dstRect: Rect): BOOLEAN;"
    <primitive: 160 trapA: 16rCAA type: 'B-RRVR'>
                "trap number"  "Boolean function, ignore receiver,  rect, rect, VAR rect"

    ^self primitiveFailed
```

```
newPtr
    "Pascal =        FUNCTION newPtr (logicalSize: Size): Ptr;"
    <primitive: 160 trapA: 16r01E type: 'D2-L0'>
                "trap number"  "pointer function result in A0, ignore receiver,
                                longint input param in D0"

    ^self primitiveFailed
```

The conversion routines check the type of each argument passed on the Smalltalk stack against the **Type Descriptor** information in the call to primitive 160 or 162. A primitive failure is flagged if there is a type mismatch. The interpreter routine CallTraps then converts each value appropriately and calls the designated trap.

Consider the case of SetPt (VAR pt: Point; h, v: INTEGER) which could become:

```
setPt: pt h: h v: v
    "Pascal =        PROCEDURE SetPt (VAR pt: Point; h,v: INTEGER);"
    <primitive: 160 trapA: 16rC80 type: '--VPII'>
                "trap number"  "no function, ignore receiver,  VAR point, integer, integer"
    ^self primitiveFailed
```

In this case point does not need to have valid field values on input. The Smalltalk object passed in must be a Smalltalk point. It would be OK here though if this point

were created by the Smalltalk statement 'Point new'. This statement creates a point whose fields are both NIL. When they get Var parameters with NIL fields, the conversion routines pass zero as the input values to the Toolbox. The return values from the Toolbox call are then copied into the fields of the point parameter.

The Toolbox routine setPt could optionally be encoded as:

```
"point" setx: x y: y
    <primitive: 160 trapA: 16rC80 type: '-VPII'>
                    "trap number" "no function, receiver is VAR point, integer, integer"
    ^self primitiveFailed
```

Here the receiver (hence the comment "point") is also used as the first parameter so we save pushing a parameter on the stack.

SmallIntegers and Booleans are not allowed as VAR parameters. For LargeIntegers and Strings as VAR parameters, the input object is converted into the result object via a Become. For Points, Rectangles, and PascalRecords, the new values for the appropriate pointer fields are substituted within the original object.

When using data in this Smalltalk/Toolbox environment, you must not pass the Toolbox the address (pointer or handle) of any data stored in the Smalltalk heap if the Toolbox is going to remember that address after it returns from that one call. It is not possible for the location of a Smalltalk ByteArray object to change during the time a single Toolbox call is running. It can change between calls, however; so ByteArrays should only be used for data whose address is passed to the Toolbox every time the Toolbox uses the data. This way the conversion routines will create the correct address each time.

In both primitives, the 'primitive: <160 or 161>' part is optional. For example, primitive 160 can be called by

<trapA: 16rC71 type: '--VP'>

and primitive 161 can be called by

<recordOffset: 16 type: 'I'>.

Appendix 3
A Summary of Useful Expressions from the Level 1 System Workspace

Changes

Smalltalk noChanges.
Resets the set of changes to be empty.

Form removeFromChanges.
Removes all changes in class Form from the changes set.

Smalltalk changes asSortedCollection
printIt to see a list of all changes in the changes set.

Smalltalk browseChangedMessages.
Opens a message browser on all methods in the changes set.

(FileStream fileNamed: 'Changes.st') fileOutChanges.
Writes a file of all changes in the changes set.

FileStream fileNamed: 'PenChanges.st') fileOutChangesFor: Pen.
Writes a file of all changes for class Pen in the changes set.

(FileStream oldFileNamed: ' Changes.st') fileIn.
Reads in changes from the file 'Changes.st'

Files

(FileStream fileNamed: 'Hello') edit.
Opens a file edit window on the file 'hello'.

FileDirectory filesMatching: '*.st'
*PrintIt to view file names. It is actually more convenient to open a file list and type *.st<Return> in the top pane.*

Inquiry

InputState browseAllAccessesTo: 'deltaTime'.
Like 'inst var refs' in the browser.

Smalltalk browseAllCallsOn: #isEmpty.
Like 'senders' in the browser.

Smalltalk browseAllCallsOn: #showWhile: and: #read.
Browses methods which send both these messages.

Smalltalk browseAllImplementorsOf: #includes:
Like 'implementors' in the browser.

Smalltalk browseAllCallsOn: (Smalltalk associationAt: #Mac).
Browse all references to a global variable

Smalltalk browseAllCallsOn: (Cursor classPool associationAt: #ReadCursor).
Browse all references to a class variable

Smalltalk browseAllCallsOn: (Undeclared associationAt: #Disk)
Browse all references to an undeclared variable

Smalltalk browseAllMethodsInCategory: #examples
The name says it all.

Smalltalk browseAllSelect: [:m | m isQuick].
(hackers) Browse all methods for which the block is true.

Smalltalk browse: Pen
Open a browser on the class Pen

Enumeration

Smalltalk printSpaceAnalysis

Creates a text file listing number of objects and words of space used by the code of each class and by the instances of that class.

Smalltalk garbageCollect.

Force Smalltalk to do a full garbage collection. Use before the following enumerations...

FileStream instanceCount

PrintIt to find out how many there are currently

StrikeFont allInstances inspect.

Inspect an array of all instances of this class.

(Smalltalk collectPointersTo: StrikeFont someInstance) inspect.

Opens an inspector on all objects which point to one of the fonts.

Undeclared Variables Dictionary

Undeclared keys

printIt to see names of any undeclared variables.

Undeclared inspect

Nicer way to view undeclared variables if you're in Level 1 or if you've filed in DictionaryInspectors

Undeclared ← Dictionary new.

Resets the dictionary of undeclared variables. Note this will preclude your being able to find the methods in which they occur.

Undeclared associationsDo: [:assn | Smalltalk browseAllCallsOn: assn]

Browses all references to the first undeclared variable encountered.

Dependents Dictionary

(Object classPool at: #DependentsFields) keys

printIt to see what is in the global dependents dictionary

(Object classPool at: #DependentsFields) keysDo:
[:each | (each isKindOf: DisplayText) ifTrue: [each release]]

Releases certain objects from the dependents dictionary.

Housekeeping

Transcript clear.

Resets the transcript window.

Smalltalk allBehaviorsDo:
[:class | class removeSelector: #DoIt; removeSelector: #DoItIn:].

Removes methods generated by doIt and printIt.

Smalltalk removeKey: #GlobalName.

Removes a symbol from the global dictionary. You should check first that there are no outstanding references to it.

Smalltalk declare: #GlobalName from: Undeclared.

Will make GlobalName be a new global variable. If it was formerly in Undelcared, then all references to it will be forwarded to the new global variable. This is how forward references are handled in fileIns.

Source Files

SourceFiles ← Array
with: (FileStream oldFileNamed: Smalltalk sourcesName) readOnly
with: (FileStream oldFileNamed: Smalltalk changesName).

Establishes source and changes files.

SourceFiles do: [:x | x close].

Closes source and changes files.

SourceFiles ← Array new: 2.

Causes Smalltalk to run with no source or changes files (although retrieve.st will still work).

Smalltalk condenseChanges

Copy all methods in the changes file to a new file, and then replace the current changes file by the new one. In the process, multiple copies of method definitions and other unnecessary text is removed. This process may take a long time, and will require sufficient disk space for both copies of the file during the process. You should make backup copies of your image and changes before running this method.

Measurements

Smalltalk spaceLeft

printIt to see how much space is currently available.

Symbol instanceCount

printIt to see how many Symbols are in use

Symbol rehash.

Purge any Symbols which are no longer in use.

Time millisecondsToRun: [Pen example]

printIt to see how long it takes to execute the block.

MessageTally spyOn: [Pen example].

(Level 1 only) Presents an time-use profile for execution of the block.

Crash recovery

Smalltalk recover: 5000.

Opens a window on the last 5000 characters added to the changes file

ScheduledControllers removeAndUnschedule: <someController>

Execute this from a debugger if a window gets scheduled which is incapable of responding properly to all window protocol. You'll have to poke around until you find the top-level controller, and then type an expression that refers to it in place of <someController>