A Debugger for the PostScript Language*

Michael A. Harrison
Fred Meyer
Computer Science Division
University of California
Berkeley, CA 94720

April 29, 1991

Abstract

This report describes dbps, a PostScript[1] debugger which runs under the X11 Window System. dbps provides the user with features which allow the user to do more than display the output of their programs at their workstation. dbps enables the user to observe the state of the interpreter as the program is executing, suspend the execution of the program, and interact with the interpreter before, during and after the execution of the program. The design and implementation of dbps are discussed in this paper. The motivation for dbps is provided by an overview of the PostScript language and how the interpreter handles errors along with a survey of current PostScript previewers.

1 Introduction

PostScript has emerged as the standard for graphic page description. This page description language is device independent and contains powerful built-in graphics

*Sponsored by the Defense Advanced Research Projects Agency (DARPA), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292. This report is an edited version of the Masters of Science thesis of the second author written under the supervision of the first.

[1]PostScript is a trademark of Adobe Systems Incorporated.

commands. Programs in PostSript are sometimes written by an individual, but for the most part are generated by an application's printer driver to produce output on a PostScript-driven raster output device (traditionally, a laser printer). Creating PostScript programs and correcting errors are not always easy. When the PostScript interpreter inside the laser printer encounters an error, it quits executing the program and sends an error message back to the connected host machine or to a printer log. In many system configurations this error message is ignored and lost. Therefore, a PostScript program with an error sent to a laser printer produces literally nothing.

To avoid the difficulties and inconveniences of sending a PostScript program to the printer when a paper copy is not needed, several PostScript previewers have been developed to run on graphic workstations[5, 4, 3, 6]. The previewers will interpret a PostScript program and display the output to the screen of the workstation. These previewers provide a convenient method to view the output of a program and will indicate what errors if any occur. However, they do little to display the state of the interpreter which would assist the user in locating and correcting an error.

This report describes dbps, a PostScript debugger, developed under the X11 window environment, which provides PostScript program designers an easy to use and informative user interface to a PostScript interpreter. The features provided by the interface enable the program designer to observe the state of the interpreter as the program is executing, suspend the execution of the program, and interact with the interpreter before, during and after the execution of the program.

The next sections will describe the PostScript language, how programs are created and how errors are handled. This is followed by a discussion of existing PostScript previewers. The design and implementation of dbps are presented in the following two sections. The final section presents some conclusions and suggestions for future work.

2 PostScript

PostScript is a computer programming language developed by Adobe Systems, Incorporated to communicate high-level graphics information to digital laser printers. Because of its powerful graphics commands, PostScript has become a standard as a page description language. These commands can describe text, graphics and sampled images and each is given uniform treatment. This greatly facilitates an application's ability to combine all of these elements on the same page.

PostScript can be described not only as a page description language but also as a general purpose programming language. It contains powerful built-in graphics commands which makes it well-suited as a high-level device-independent interface between a composition application and a raster output device. The composition application, such as a word processor or computer-aided design application, produces output on a raster device in a two-stage process. First, it generates the PostScript page description and sends it to the raster output device. The output device is controlled by the PostScript interpreter which takes the high-level page description and converts it to a low-level raster data format for that specific device. With the page description language as the intermediary, the composition application is able to submit its output to any PostScript driven raster output device.

As a general-purpose programming language, PostScript has syntax, data types and execution semantics. These elements are a part of any PostScript program whether or not it constitutes a page description. There are three main aspects of the PostScript language: it is interpreted, stack-based and uses an unique data structure called a dictionary. The dictionary mechanism gives PostScript a flexible and extensible base. New PostScript operators can be defined, using existing operators, and then stored in dictionaries for later use in the program. PostScript is a stack-based language similar to that of FORTH[2]. It uses a postfix notation in which the operands are placed on the stack and later consumed by operators. PostScript is also an interpreted language in which each program is created, transmitted and executed as source text. There is no compiling or encoding of the program and it need not run directly on a CPU. An interpreter process reads the input sequence of characters and breaks it into objects according to the PostScript language semantics. These objects are then manipulated to change the current state of the PostScript environment.

2.1 Writing PostScript

Most PostScript programs are not written by an individual. This is not to say that a program cannot be constructed from scratch, but this is not the main intention of the PostScript language. PostScript programs are intended to be machine-generated. That is, other application programs such as document processors use PostScript to express complex graphics in a device-independent format. The part of the application program that produces the PostScript page description is called the device driver [2]. Part of the printer driver, the script, is generated by the application software and part,

[2] The principal device is usually a printer but both Display PostScript and NeWS incorporate PostScript into the window system

the prologue, is written by the application designer. The prologue defines operators and sets up the PostScript environment to suit the need of the particular application. The script contains the actual information which is to be displayed on the page.

The application designer uses the printer driver to set up methods for adapting the application's graphic data structures to that of the PostScript language image model. The designer must consider what portion of the PostScript program should be machine-generated or written by hand. For efficiency, the script should be compact. This is accomplished by defining procedures in the prologue which are executed in the script to perform more complex tasks. Creating these procedures may take experimentation to find the most accurate and efficient method for translating between the two graphic structures. Modularity in PostScript programs is also an important aspect of program design as it is with any programming language. It is strongly recommended that each page be independent of the others and that the page elements on the same page be independent with well-defined interfaces between them. By adhering to these guidelines, the application developer can create and test each aspect of transformation between graphic states. Some guidelines for PostScript programming style are given in [8].

2.2 PostScript Errors

Errors in PostScript can be put into two categories: execution errors and "off the page" errors. Execution errors are ones in which the rules of the PostScript language have been violated. The interpreter is unable to continue executing the failed program and flushes it. "Off the page" errors are ones in which the coordinate system mapping has been modified in such a way that the image is painted off the page. This is not an error to the interpreter but is likely an error to the user because the image is not on the page.

PostScript programs are executed by the interpreter inside a job server loop. This loop executes the user program inside of a stopped context[3] in order to regain control if an error occurs. When an error occurs, a procedure is executed for that particular error. The following steps are taken when an execution error occurs:

- The interpreter finds the procedure for the specific error in the dictionary `errordict` and executes it.

- The procedure captures the contents of the operand, execution and dictionary

[3] {...user program...} `stopped`, if an error occurs `stopped` returns a true otherwise false.

stacks and places them into an array in a sub-directory of `errordict` known as `$error`.

- The name of the error is stored under `errorname` in `$error` along with the offending command under the name `command`.

- Finally, the procedure executes the `stop` command.

Whenever `stop` is executed, control returns to the innermost `stopped` context. If the server loop `stopped` command returns the value `true`, it executes the procedure `handleerror`. This procedure sends an error message back to the host computer.

Even with PostScript's many error handling mechanisms, it is often difficult to determine the specific problem with a program. The PostScript device (i.e, the printer) is often shared on a network. This configuration poses two problems. First, because the device is shared it is not possible for one person to take sole control of it to debug their program. Also, direct communication with the device may be difficult to setup. Messages sent back by the PostScript device may not reach the sender of the job. These messages may be logged in a printer log or displayed on the host terminal but often the messages are ignored and lost. When the sender does receive the message, it does not always provide enough information to find the error.

2.3 Testing and Debugging

Testing and debugging an interpreted language should be fast and easy. Because an interpreted language is not compiled, there are no steps between the creation of the file and execution. Postscript is interpreted, but it can still be difficult to test and debug a PostScript program. Complications arise from the fact that the interpreter may run on a remote processor over which the developer has little to no control. In many environments it is difficult or even impossible to set up interactive communications with the PostScript device. To address this problem many PostScript previewers have been developed which run on graphic display workstations. These previewers give application designers the ability to execute PostScript programs and view the created image on the display of their workstation. Although they do not provide the designer with interactive communication to the interpreter, the user is able to execute their programs to discover if and when an error occurs. If the program is not correct, the designer can quickly make a change and retry the program. Aside from the quick response time, PostScript previewers do not provide the designers with any more information than they would have received from the printer.

3 PostScript Previewers and Debuggers

GhostScript, dxpsview, UCBPS and x11ps are a few of the existing PostScript interpreters and previewers which run on graphic work stations under the X window system. LaserTalk[1] is a PostScript debugger which runs on the Macintosh[4] and uses the LaserWriter[5] as an interpreter. Each of the previewers allows the user to display the output of their PostScript programs on the display of their workstation. However, each of these applications function differently and offer the user a variety of services.

3.1 GhostScript

GhostScript is a previewer/interpreter for the GhostScript language. The GhostScript language, developed by Aladdin Enterprises, bears a very strong resemblance to the PostScript language. It conforms, almost exactly to version 25.0 of the PostScript language description as described in the PostScript language reference manual[7]. GhostScript also extends the PostScript language by adding additional operators.

The user invokes GhostScript from the UNIX command line by entering: `gs <filename1> ... <filenameN>`. The interpreter reads the files in sequence and executes them. The images created are displayed in a new window which GhostScript brings up. Following the execution of the files, the user is presented with a `GS>` prompt in the originating terminal window. At this prompt the user may enter additional GhostScript commands one line at a time. To exit the interpreter, the user may enter the GhostScript command `quit`. An end-of-file character or an interrupt character can also be used to terminate the interpreter.

When the GhostScript interpreter encounters an error, it provides the user with an error message in their terminal window. This error message contains the name of the error, the offending command and the contents of the operand, execution, and dictionary stacks. This error message is difficult to use because the execution and dictionary stacks are presented in a fairly cryptic form. The presentation of the dictionary stack is not a list of names as might be expected. Instead, it is a list of number pairs depicting the size and usage of each dictionary (e.g, 35/100, 35 elements defined out of 100 possible). The execution stack contains a list of nontrivial strings which for the most part have to do with the GhostScript executive and nothing to do with the terminated user program.

[4]Macintosh is a trademark of Apple Computer, Incorporated.
[5]LaserWriter is a trademark of Apple Computer, Incorporated.

Following the error message, the user is presented with the `GS>` prompt. At the prompt the user may enter GhostScript commands to explore the state of the interpreter. This ability to do further exploration of the interpreter at the time of the error is a great advantage in determining the exact nature of the error and how to correct it.

3.2 Dxpsview

Dxpsview is a PostScript previewer which comes with DECstations running the X window system. It consists of an interpreter written by Adobe Systems and an X11 window system implementation created by DEC. Dxpsview allows the user to display the output of their PostScript programs in a window at their workstation. Because the interpreter was written by Adobe Systems, the image produced is exactly as it would appear if the program was sent to a PostScript printer.

Dxpsview is started by entering `dxpsview <filename>` at the UNIX command line. Programs are sent to the interpreter from the command line when starting dxpsview or from a file selection dialogue when dxpsview is running. The image is displayed in a viewer window which contains scroll bars to move the window around the image. The display also contains command buttons to allow the user to view the next and previous pages of the programs output if they exist. This feature makes dxpsview well-suited for previewing multi-page documents before sending them to the printer.

Dxpsview is strictly a previewer in that the user is not given the opportunity to communicate with the interpreter in any way. Dxpsview executes the user's programs in a batch mode. This means the user's programs are sent to the interpreter one by one. If an error occurs in a program, the user is informed of the type of error and the offending command. The user is not given any other information. Also, because dxpsview is strictly a previewer, the user is not given the opportunity to explore the interpreter's state at the time of the error. The program is then flushed and dxpsview is ready to accept another program.

3.3 UCBPS and x11ps

UCBPS is an interpreter which faithfully implements almost all of the PostScript interpreter language[6]. The PostScript interpreter was developed at the University of

[6] Only the image operator is not implemented.

California at Berkeley[7]. Users execute UCBPS from the UNIX command line and communicate with it over the standard I/O channels. It is used to execute PostScript programs and report errors back to the user's terminal. UCBPS also has an executive procedure which provides the user with an interactive mode. In the interactive mode the user is given a prompt at which to enter PostScript commands. The commands are executed and any output to standard output or standard error is reported to the user's terminal. Successful execution is indicated by the return of the prompt. If an error occurs the error message reported is the same as that of the Apple LaserWriter, stating what error occurred and the command that caused the error.

x11ps is a PostScript previewer also developed at the University of California, Berkeley. As its name suggests, it is a previewer which runs under the X window system. x11ps is actually a shell script which starts a special version of the UCBPS interpreter. This version of UCBPS contains a special dictionary (X11dict) which contains PostScript operators for creating and maintaining an X window. The x11ps script also defines additional operators in the X11dict and creates a window to display the user's program.

The user starts x11ps by entering x11ps <filename> at the UNIX command line. A new X window is created and the PostScript image is displayed. If no filename is given as an argument, x11ps creates an empty window. Additional PostScript programs can be executed by typing an "`f`" in the window. The user is then prompted for the name of the program to be executed. The same program can be quickly executed again by typing "`r`" in the window. This allows the user to make a change to the program and quickly send it to the interpreter.

x11ps has additional commands which are bound to keys that are typed in the X window. The user is able to evaluate a PostScript statement by typing an "`e`". At the `eval:` prompt the user can enter the statement. A carriage return sends the statement to the interpreter. The current contents of the operand stack can be viewed by typing an "`s`" in the X window. A popup dialogue displays the contents of the stack. Because only a portion of the image is displayed in the window, the keys "`l`", "`r`", "`u`", "`d`", "`t`", "`b`", "`h`" are bound to commands which move the window left, right, up, down, top, bottom, and home.

When an error occurs, the user is notified by a popup dialogue. This dialogue contains the error which occurred and the offending command. Along with this

[7] John Coker wrote most of the first version. Steve Procter contributed extensively to the graphics aspect, especially the algorithms for clip and fill. Doris Karlson Tonne ported the system to X11 and completed the interpreter.

information the contents of the operand stack are also displayed in the dialogue. The user must then click the mouse button in the dialogue to pop it down and continue further communication with the previewer. Once the dialogue is popped down the current program is flushed and the state of the interpreter is reset. This makes it impossible to do any further investigation of the interpreter's state.

3.4 LaserTalk

LaserTalk is a PostScript language development environment created by Emerald City Software to run on the Macintosh. More than a PostScript previewer, it provides users with an environment to create and debug their PostScript programs. The interpreter used by LaserTalk is inside of the LaserWriter printer. LaserTalk communicates with the Apple LaserWriter over the AppleTalk[8] network.

LaserTalk operates in two modes, "online", connected to the LaserWriter or "offline", not connected. When not connected, the user can use the file Editor/Debugger window to create and edit PostScript programs. Also, while offline, the file in the editor can also be downloaded to the printer for printing. Connecting with the printer starts an interactive conversation between LaserTalk and the LaserWriter. LaserTalk loads a special dictionary into the interpreter which contains operators LaserTalk uses to retrieve information about the interpreter. LaserTalk also starts an executive in the interpreter and presents the user with an Interactive window. In this window, the user can communicate interactively with the interpreter. The user is also able to display various elements of the interpreter's context in the Status window, explore dictionaries with the Dictionary Browser and display the created image in the Previewer window. In the Editor/Debugger window the user is now able to send a program line by line to the interpreter and observe the changes in its state.

When an error occurs while debugging a program or during conversation in the Interactive window, a dialogue box is popped up indicating the error. This error message contains the type of error which occurred and the offending command. To continue, the user clicks on the "Ok" button in the dialogue. At this point the interpreter is in the same state as it was before the execution of the offending command. The user can easily investigate the error by examining the contents of the Status window or by a dialogue with the interpreter in the Interactive window. If the error occurred while stepping through a program, the erroneous line may be corrected and sent without having to resubmit the entire program. Also, because the status

[8]AppleTalk is a trademark of Apple Computer Incorporated.

window is being continually updated, the user has the ability to observe the changes in the interpreter's state just before the error occurred. This can give the user some important clues as to the cause of the error.

4 dbps Design Goals

dbps has been designed as a tool to make the creation and debugging of PostScript programs easier and faster. Anyone who has written PostScript programs knows how difficult it is to get them to produce the desired effects. It requires many iterations of trial and error before the output is correct. This trial and error process can be slow and tedious (not to mention a waste of paper) if each attempt must be sent to the laser printer with the hope that something comes out. The research described in this report involved the design and implementation of a tool to overcome the problem described above.

dbps is designed to provide the user with an environment to create, debug, and experiment with PostScript programs. The desired features of this environment are:

- Interactive communication with the PostScript Interpreter.
- A prompt display of the current PostScript context.
- The display of the current page.
- The ability to step through a PostScript program and observe the effects of each statement.
- The capability of browsing through commands defined in dictionaries.

This section describes the design issues for each of these features.

4.1 Interactive Communication

It is important for dbps to preserve the ability of the user to communicate with the interpreter directly, that is, to be able to carry on an interactive dialogue with the interpreter. This will not only assist the user in debugging a program but will also give the user the opportunity to experiment.

When debugging a program, the user needs the ability to modify the PostScript context to determine exactly how to correct a problem. By allowing the user to create commands specific to the program, these commands can then be used as tools to assist in the debugging. For example, in debugging a procedure the user can create a command to reset the state of the interpreter to the state that existed before the execution of the procedure. This provides a convenient way for quick retries.

Designers of PostScript programs may wish to experiment with the language during their program development process. Persons unfamiliar with the language would also want to experiment with and learn how PostScript's built-in commands work. Novices and experts can also develop and test procedures before entering them into their programs. This ability to work out bugs and get commands working correctly before entering them into the body of a program is very desirable.

Previewers such as GhostScript and LaserTalk provide the user with the above functionality. These applications are examples of existing tools which demonstrate the usefulness of an interactive environment and provided the inspiration to include at least this functionality in dbps.

4.2 Display of the PostScript Context

A natural extension of being able to experiment interactively, is the ability to monitor the current state of the interpreter. The results of every command depend on and change the context of the interpreter. The user, being able to monitor the context, is better able to understand errors and detect when they occur.

In direct communication with the PostScript device, context information can be requested by the user. The interpreter returns the information back to the user's terminal which is connected to the interpreter's standard output. For example, the `pstack` command will non-destructively print the contents of the operand stack to standard output. Not all aspects of the PostScript context are this easy to get. The stack is the only part of the context which has a built-in command to print it to standard output. The difficulty in printing other aspects ranges in complexity from a few instructions to complex procedures.

LaserTalk's status window is an example of the usefulness and convenience of having the context information promptly and automatically displayed. Its usefulness has demonstrated the need for a similar functionality in dbps. However, LaserTalk's implementation of the status window can be awkward. Everything is displayed in one window in whatever order the user has selected. Because of this awkwardness, dbps separates the status window into multiple windows.

Some of the elements in the PostScript context are important for all aspects of debugging and experimentation. For this reason, they should be continuously displayed in their own window. For example, the operand and dictionary stacks are two of the most important elements of the PostScript context. These stacks are involved in the execution of every command. Therefore, it would be beneficial for debugging and experimentation if these stacks are constantly displayed in their own windows.

Other elements vary in importance depending on the goals of the particular application. Because it is not necessary for them to be displayed continuously, they should be displayed in a list whose contents can be chosen by the user. By displaying this list in alphabetical order, finding a particular context element would be greatly facilitated.

4.3 Current Page

The purpose of a PostScript program is to produce an image on a piece of paper. The need for debugging a program may not only be to find execution errors but also to correct the appearance of the image on the paper. While debugging a program, the user must be able to see the image as it is being created. Therefore, dbps needs to provide the user with a previewer window which contains the image.

LaserTalk handles the display and management of its previewer window in an unique manner. It is able to download the contents of the LaserWriter's frame device to create an image on the Macintosh's screen. Getting the LaserWriter to dump the contents of its frame device to standard output is not one of its normal operations. The interpreters and previewers which run on graphic workstations usually do not have this ability. Because dbps is designed to communicate with an interpreter only over standard I/O channels, the creating and maintenance of the previewer window must be handled by the interpreter and not dbps.

4.4 File Debugger

When the interpreter encounters an execution error it returns a message stating what error occurred and the command that caused the error. Even if the user should receive this message, it may not contain enough information to correct the problem. Non-execution errors can be even more difficult. This is because the only information the user receives is an incorrect image. A user with an error in a PostScript program would like to be able to step through the execution of the program and observe the effects that the execution of each statement has on the interpreter's context. This ability would allow the user to quickly pinpoint the cause of the error. Monitoring the changes in the interpreter's context would also give the user clues as to how to fix the error.

The above difficulties apply not only to PostScript printers. Many previewer tools such as dxpsview, GhostScript, and x11ps suffer from the same problems. These applications provide the user with the ability to execute their programs at a graphic workstation and to quickly see the results. If an execution error does occur, the user always receives the error message that might have been lost from the printer. However, the viewers do not give any detail of the events which led to the error. These details are important for the user to find and correct the error.

LaserTalk provides the user with the ability to step through a program. This allows the user to monitor the PostScript context and better determine the events which lead to an error. LaserTalk has an easy to use graphical interface for setting break points and sending statements to the printer. However, LaserTalk does not provide the user the ability to set a break point inside a procedure. This makes the debugging of procedures difficult because the user does not have the ability to stop the execution of the procedure to examine portions of the PostScript context.

Based on the above issues, one of dbps's goals is to provide the user with an environment in which he/she can step through a PostScript file and monitor the context of the interpreter. Experience with Laser Talk has also demonstrated the need for dbps to provide an environment that permits break points to be set inside and outside of procedures.

4.5 Dictionary Browser

The PostScript language contains an unique data structure called the dictionary. Everything currently defined in the PostScript language is defined in dictionaries. There are times when the user may wish to look at what is currently defined. The user may be debugging a program because he/she has received an undefined error for an operator defined earlier in the program. By searching the dictionaries which are currently defined at the time of the error, the user may find that the operand is no longer defined or has been misspelled. He/She could also discover that the operand is defined in a dictionary which is not currently on the dictionary stack.

When a user is experimenting or writing a program, he/she may wish to see a list of available commands. For example, the user may wish to see the names of the available fonts. By listing the FontDirectory (which is a dictionary), the user can see a list of font names which are currently available.

The PostScript code to list the keys of a dictionary is very simple: *dictionary* `{pop`

`=} forall`. The above code will list the keys of the dictionary to standard output. However, the user must be able to interactively communicate with the interpreter at the time the search is desired (i.e., at the time of an error). Because some of the keys may represent other dictionaries, the search procedure can become complicated. In addition, entering this statement for each dictionary to be searched is tedious. dbps needs to provide the user with an interface which allows the user to easily list the keys of currently defined dictionaries. The list of keys need to be in alphabetical order and must also provide the user with a display of each dictionary contained therein, so that the user is able to search every dictionary.

The user may wish to find where a particular command or name is located. If the user wants to execute a particular command, the dictionary which holds its definition must be on the dictionary stack.

In PostScript, it is easy to accidentally or intentionally rename an operator. This occurs when an operator is defined by giving it the same name as one lower on the dictionary stack. When the interpreter searches for names it finds the one defined higher on the stack. Even PostScript built-ins can be redefined in this manner. When the execution of an operator does not perform the expected operation, the user may wish to look at its definition. By examining its definition he/she can tell if this is the intended operator.

To allow the user to view the definitions of all the dictionaries on the dictionary stack, a goal of dbps is to provide the user with a tool to investigate the contents of currently defined dictionaries. LaserTalk's dictionary browser demonstrates that a useful and intuitive graphical interface would allow the user to easily and coherently perform this search. By including a similar feature in dbps, the user would be able to:

- Display all the dictionaries currently defined.
- Display a list of keys defined in each dictionary.
- Display the associated value of a particular key.

It would also be beneficial to be able to browse the dictionaries in the middle of debugging a file or when an error occurs.

5 Implementation

This section discusses the implementation of dbps whose design goals were presented in the previous section. dbps was implemented using the Athena Widget Set[10] and the X Toolkit Intrinsics[9] as an interface to the UCBPS interpreter.

5.1 Communication

dbps is implemented as a separate interface process to that of UCBPS. It communicates with UCBPS via pipes connected to the interpreter's standard I/O channels. dbps sends PostScript commands to UCBPS's standard input and reads from its standard output. For each command sent, dbps reads UCBPS's standard output until there is no more to read. A special protocol and executive are used for the communication between the two processes.

The user starts dbps from the UNIX command line. The dbps process starts UCBPS by forking and execing. The communication channels are then set up and the initialization file db.ps is sent to UCBPS. This file contains the protocol information and also creates PostScript commands to be used by dbps. The final command in the initialization file starts dbps's special executive procedure. This procedure handles all further transactions between UCBPS and dbps. Finally, the user is presented with dbps's main window as shown in Figure 1.

dbps communicates with UCBPS by sending it PostScript commands and reading the interpreter's output line by line. This communication is facilitated by a simple signaling protocol. dbps appends code to the end of every Postscript command it sends. This code instructs UCBPS to send an "end of transmission" signal to dbps at the end of execution of the user's command. This signal is a nonprintable character in the first position of a line read from UCBPS. When dbps receives the "end of transmission" signal, it knows that UCBPS is ready for another PostScript command.

dbps also looks for other signals in this protocol. These signals come to dbps asynchronously and are also in the first position of the line. These asynchronous signals are:

- SIG_BEGIN_START
- SIG_BEGIN_END
- SIG_END_START
- SIG_ERROR_START

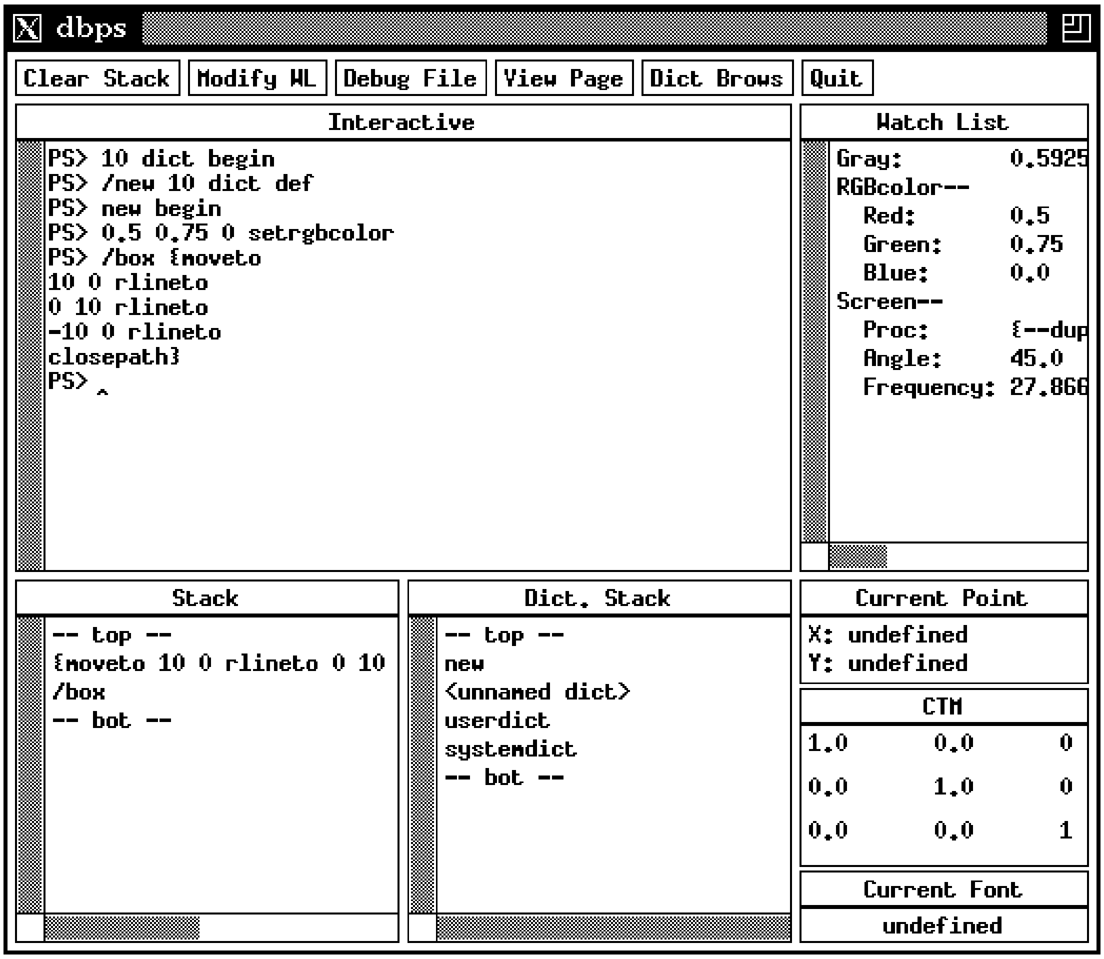

Figure 1: dbps's Main Window

- SIG_ERROR_END
- SIG_BREAK_START

SIG_BEGIN_START and SIG_BEGIN_END signals are sent when the `begin` command is executed. The `begin` command has been redefined in `userdict` to send dbps the name of the last dictionary pushed onto the dictionary stack. dbps will find the name between these two signals.

The SIG_END_START signal is sent whenever the `end` command is executed. The `end` command has also been redefined to tell dbps that a dictionary has been popped off the dictionary stack.

SIG_ERROR_START and SIG_ERROR_END signals are sent when an error occurs. They are sent from inside the executive's error procedure. dbps is able to find information about the error (i.e, the name of the error and offending command) between the two signals.

dbps communicates with dbExecutive, a special executive running in UCBPS. dbExecutive has been written to handle the communications between dbps and the interpreter. Because dbps will break inside a PostScript procedure when debugging a file, this executive is more complicated than UCBPS's default executive. Also, because UCBPS allows only a limited number of curly braces to be open in a procedure definition at one time, dbExecutive needed to be defined in two procedures:

```
/dbInterLoop {
   {dbps /dbHoldFile get token
        { {dup type (arraytype) eq not {exec} if}
          stopped
          {$error /newerror get
              { /dbERROR_START $db
                errordict /handleerror get exec
                (\n) = /dbERROR_END $db /dbEND $db exit
              }
              {stop}
              ifelse
          }
          if
        }
        {exit}
        ifelse
```

```
   } loop
} bind def

/dbExecutive {
   { flush
       {disableinterrupts (%statementedit) (r) file enableinterrupts}
       stopped
       {pop pop $error /newerror false put enableinterrupts}
       {dup status not
            {pop (quit\n) print flush exit} if
            dbps exch /dbHoldFile exch put
            {/dbInterLoop $db} stopped {exit} if
       }
       ifelse
   } loop
} bind def
```

5.2 Interactive Window

The Interactive window to the PostScript interpreter has been implemented as a subwindow of the main window in Figure 1. This subwindow is a text widget which prompts the user to enter PostScript commands. When a complete command is entered, it is sent to the interpreter. If the user types a carriage return before completing a full command (e.g., an incomplete procedure definition) dbps will hold the input and not return the $PS >$ prompt until the command is completed. The cursor is then moved to the next line where the user is able to complete the command.

The text widget's translation table has been modified to call a parsing procedure whenever a carriage return is typed. This parsing procedure reads the user's input character by character, moving the characters into a buffer to be sent to the interpreter. When a complete PostScript command has been entered, the buffer is sent to the interpreter. While the parsing procedure is transferring the characters, it is also looking for the following special characters: '(', ')', '{', '}', and '%'. These special characters delimit syntactic entities such as strings, procedures and comments.

The special character '%', not inside a string, introduces a comment. Because PostScript is a line-based language, the comment consists of all characters between the '%' and the next newline character. When the parsing procedure encounters the comment character, it disregards the rest of the line. This prevents unnecessary characters from being placed in the buffer and sent to the interpreter.

The parentheses and curly brackets delimit string and procedure syntactic entities, respectively. If either of these are not complete before being sent to the interpreter, the interpreter and dbps become deadlocked.

When the parsing procedure encounters '{', it considers itself to be inside an executable array, also know as a procedure. This is noted by incrementing an internal counter. The '{' is added to the buffer and the parsing procedure continues. If additional '{'s are encountered, the counter is incremented again. When a '}' is encountered while the parser is inside an executable array, the counter is decremented. When the parsing procedure reaches the end of the user's input, it checks to see if the counter is zero. If it is not, it knows that a complete executable array has not been entered. The user's partial command is not sent to the interpreter, but is held in the buffer. The cursor is moved to the beginning of the next line in the interactive window and control is returned to the text widget. The user does not receive a prompt but can complete the command.

When the parsing procedure encounters '(', it considers itself to be inside a PostScript string. This is noted by incrementing the internal string counter. The parsing procedure adds the '(' to the buffer along with the characters that follow without any consideration. The special characters '{', '}', and '%' are also moved to the buffer without taking the above actions. This continues until the matching ')' is encountered. The ')' is added to the buffer and the string counter is decremented. If additional '('s are encountered before a ')', the string counter is incremented again. If the end of the user's input is reached and the string counter is not zero, the parsing procedure takes the same action described in the paragraph above for the executable array.

5.3 Current Page

The current page previewer is implemented by allowing UCBPS to create its own window and for dbps to monitor the events in that window. When the user wishes to preview the page, he/she selects the *View Page* button at the top of dbps's main window. Selection of this button causes dbps to send commands to UCBPS, which installs a frame device and creates an X window where further painting will take place. Once the user opens the previewer window it cannot be closed and will remain open until dbps is terminated. The previewer window is updated after the execution of every command entered in the Interactive window or when a break is reached in the File Debugger window. To update the window, dbps sends the `copypage` command to UCBPS. `Copypage` is the same as `showpage` except it does not erase the current page or change the graphics state. The user is able to move the previewer window around the image by the same commands used by x11ps. By typing an "`r`", "`l`", "`u`", "`d`", "`t`", "`b`", "`h`" in the previewer window, it is moved right, left, up, down, top, bottom and home over the image.

This implementation was made difficult because the contents of UCBPS's frame device are not accessible to dbps over standard I/O channels. This means dbps cannot extract the contents of UCBPS's frame device to display it in a window directly under its control. The solution is to have UCBPS manage its own X window. dbps sends UCBPS some of the same commands which are in the x11ps scripts to install a frame device and create an X window. The executive started by the x11ps scripts which manages the events in UCBPS's window could not be used because it does not allow UCBPS to accept input from standard input. Therefore, dbps must locate UCBPS's window and monitor for events in that window as well as its own. dbps accomplishes this by forking off a child process which looks for events in only UCBPS's window. When events occur, dbps's child process sends UCBPS PostScript commands to update its window. These commands are sent over the same channels in which dbps communicates with UCBPS.

5.4 Stack

The stack window which displays the current contents of the operand stack is implemented as a single column list widget inside a view-port widget as shown in Figure 1. The stack window is updated after the execution of every user command entered in the interactive window or when a break is reached in the Debug File window. The operand stack information is retrieved from the interpreter by sending the `dbStack` command which is defined by the db.ps file. The `dbStack` definition is: {pstack}, which prints the contents of the operand stack to standard output. Each time the stack is updated the stack list in dbps is cleared and reloaded.

5.5 Dictionary Stack

The dictionary stack window displays the names of the dictionaries currently on the interpreter's dictionary stack. This window is also implemented as a single column list widget inside a view-port in the same way as the stack window as shown in Figure 1. Complications arise because the elements saved on the dictionary stack are the dictionaries themselves and not their names. Also, dictionaries do not always have names (e.g., `10 dict begin` places an unnamed dictionary on the dictionary stack). These complications make displaying the list of names of the dictionaries more difficult than listing the elements on the operand stack.

The dictionary stack is only affected by two PostScript commands: `begin` and `end`. These commands only affect the top of the stack: `begin` pushes a dictionary on and `end` pops the top one off. Because of this, it is only necessary to update the dictionary stack window whenever a `begin` or `end` is executed. `Begin` and `end` are redefined in db.ps to send a signal to dbps when they are executed. The new definitions of `begin` and `end` are:

```
/begin {begin /dbBEGIN_START $db = flush
        currentdict countdictstack array dictstack
        {/dbDictFor $db} forall pop
        /dbBEGIN_END $db  = flush} bind def

/dbDictFor {
     {3 -1 roll dup 4 1 roll eq {= flush} {pop} ifelse} forall
} bind def

/end {end /dbEND_START $db = flush} bind def
```

Dictionary names are added to the stack window when dbps receives the SIG_BEGIN_START signal. Upon receiving this signal, dbps expects to read from the interpreter the name of the dictionary pushed onto the dictionary stack followed by the SIG_BEGIN_END signal. The redefinition of `begin` also determines the name of the dictionary pushed on the dictionary stack and prints it to standard output. The name of the dictionary in question is found by comparing it with the values of the key-value pairs of the dictionaries currently on the dictionary stack. The search starts with the `systemdict` and works its way up the dictionary stack. If a match is found, the key of the key value pair is the name of the dictionary in question. If no match is found, an unnamed dictionary was pushed onto the dictionary stack and no name is sent to dbps. If a name does not arrive before the end signal, an unnamed dictionary must have been pushed onto the dictionary stack. In this case the string "<unnamed dict>"[9] is used in place of the name.

Names are removed from the dictionary stack window when dbps receives the SIG_END_START signal. Because `end` removes only the top dictionary from the

[9]This string contains a space, which guarantees the user could not define a dictionary by this name.

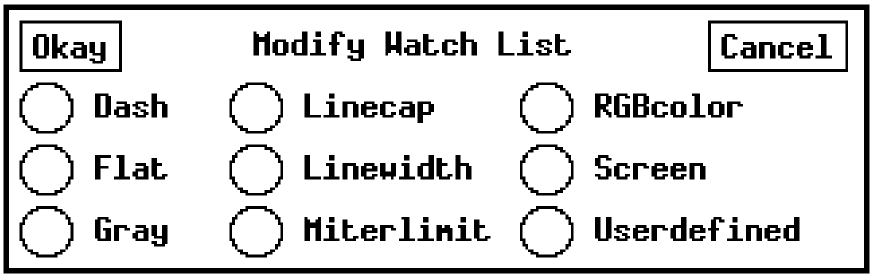

Figure 2: Watch List Selection Menu

dictionary stack, dbps needs only to be signaled whenever `end` is executed. No further information is needed to correctly maintain the dictionary stack window.

5.6 Watch List

The Watch List is a list of other PostScript context elements. The elements displayed in this list are chosen by the user. This feature has been implemented as a sub-window of the main window shown in Figure 1, along with a pop-up selection menu shown in Figure 2. The selection menu is popped up by clicking on the *Modify WL* button on the top of the main window. The user is then presented with the selection menu. To select an item to be displayed in the watch list window, the user sets the toggle button by clicking on the circle next to the name of what is to be displayed. Any number or combination can be selected. The modification is approved by clicking the *Okay* button. The *Cancel* button disregards the changes and the watch list display remains unchanged. The *Okay* and *Cancel* buttons also pop down the selection menu window.

The watch list display is implemented as a single column list widget inside of a view-port. Each selected element in the selection menu is displayed in this window. This information is updated at the same time the operand stack window is updated. dbps sends requests to the interpreter for each element to be displayed. The interpreter sends the results back to dbps by printing them to standard output. Each element of the PostScript context has different amounts of information associated with them. For example, current gray has only one piece of information and current RGB color has three. dbps expects to receive a certain amount of information per element. Each piece of information is displayed on a separate line in the watch list window.

The current point, translation matrix and current font are other context information which is continuously displayed. They are updated along with the stack window. The retrieval of the current point and font is more complicated than the retrieval of other context elements. The complication arises from the fact that both the current point and font can be undefined. When either one is undefined, the execution of the Postscript built-ins which place their values on the operand stack results in an error. Therefore, these errors need to be trapped so as not to interrupt the flow of operation. Instead of dbps receiving an error signal it reads the string "undefined" from the interpreter's standard output. The following PostScript code handles the trapping of the errors and sending of the string:

```
/dbPoint {{currentpoint}
            stopped
            {(undefined\nundefined\n) print flush
              $error /newerror false put}
            {= = flush}
            ifelse
} bind def

/dbFont {{currentfont}
          stopped
          {(undefined\n) print flush
           $error /newerror false put}
          {/FontName get = flush}
          ifelse
} bind def
```

5.7 File Debugger Implementation

The file debugger window is implemented as a separate window from the main window and is shown in Figure 3. The user starts up the file debugger by selecting the *Debug File* button from the main window. Selection of this button presents the user with a pull-down menu which has two choices: *New* and *Old*. Selection of *New* brings up the debugger window with an "Untitled" file. This feature allows the user to create and test new files. Selection of *Old* from the pull-down menu presents the user with a pop-up dialogue requesting the name of a file to be loaded into the debugger as shown in Figure 4. After entering the name of the file and pressing return or selecting *Okay*,

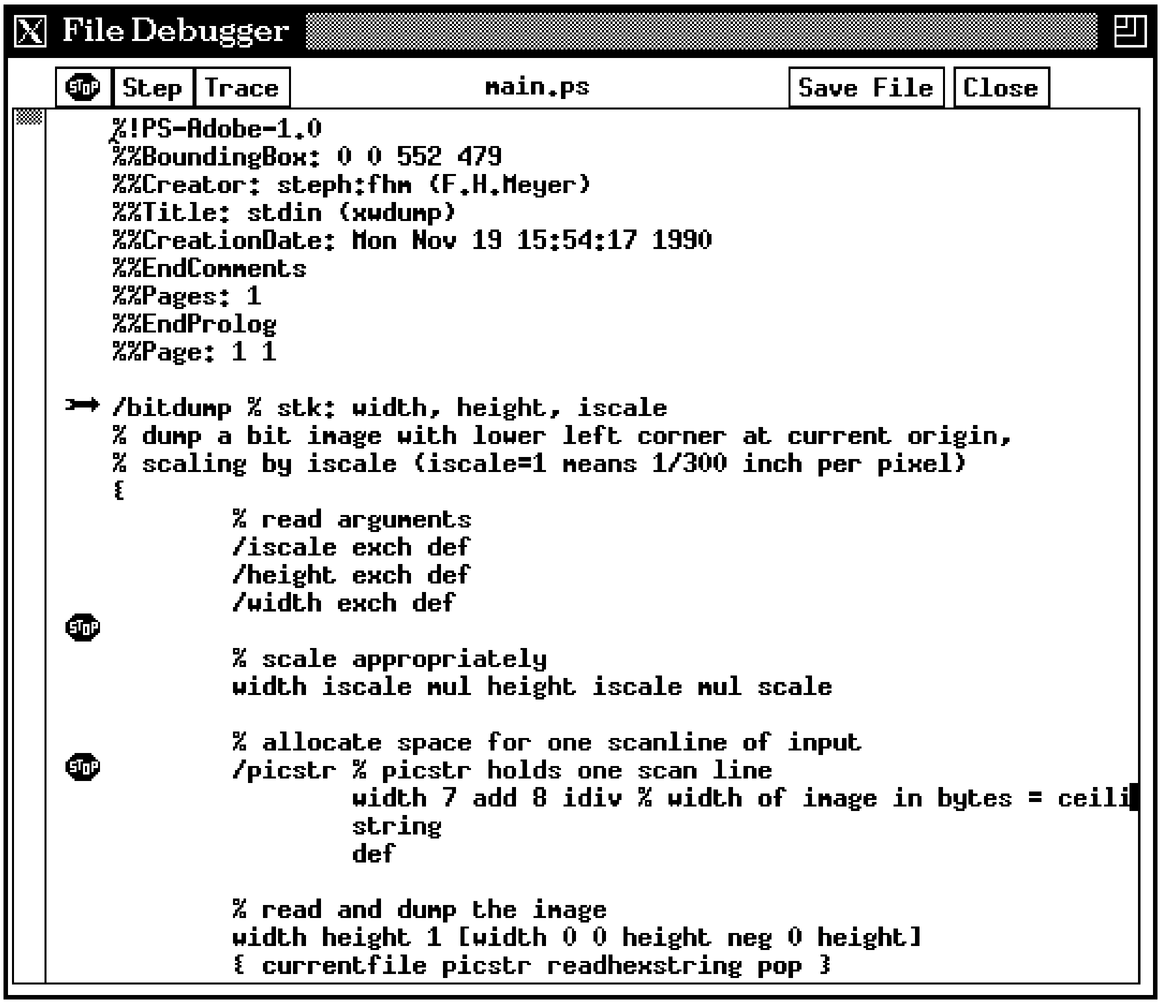

Figure 3: File Editor/Debugger Window

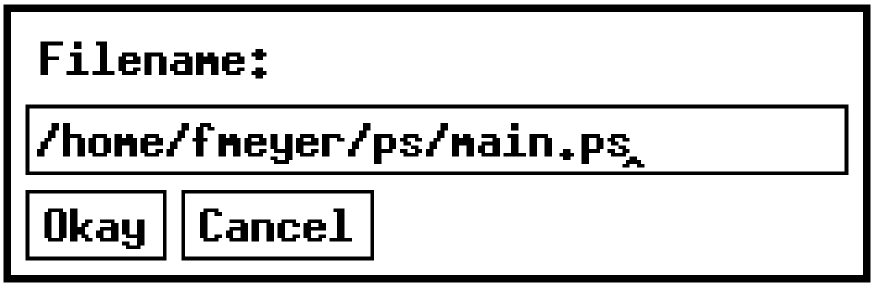

Figure 4: File Name Request Dialogue

the debugger window is brought up loaded with the requested file. Selection of the *Cancel* button aborts the file debug session.

Once a program is in the debugger the user can set break points and/or begin to send PostScript commands to the interpreter. PostScript code is taken from the text widget a line at a time starting at the current execution position, which is marked by an arrow in the left margin. Each time a line is taken the current execution position is moved to the next line. This line is examined in the same manner as if the line had been typed at the prompt in the Interactive window. If a complete command is not obtained, then more lines are taken until the command is complete. The "end of transmission" signal code is then added to the end of the command and is sent to the interpreter. dbps waits for the SIG_END_TRANS signal from the interpreter and then updates the context displays.

There are two ways the user can send commands from the file debugger to the interpreter. First, the user can step through the file by selecting the *Step* button at the top of the window. *Step* will collect one complete command from the file and send it to the interpreter. When the interpreter completes the execution of that command, dbps updates the context display and awaits further action from the user. The user may also send commands to the interpreter by selecting the *Trace* button. *Trace* will collect and send complete commands to the interpreter until reaching an outside break point (inside and outside break points are discussed below) or the end of the file. Upon reaching either one of these, dbps updates the context displays and returns control to the user.

Break points are set by selecting and moving stop signs in the margin at the beginning of a line. When the user moves the mouse pointer into the left margin of the text widget, it will change to either a stop sign or an arrow. The pointer indicates which mode the user is in. If the pointer is an arrow, clicking the left mouse button places the current execution point at the beginning of the line pointed to by the arrow. If the pointer is a stop sign, clicking the left mouse button places a break point at the beginning of that line. Switching the pointer between the stop sign and the arrow is done by clicking the *Pointer Mode* button at the top of the left margin. This button displays the mode the pointer will change to when the button is selected.

Break points can be placed at the beginning of any line. The manner in which the break is handled depends on whether it is inside or outside of a string or procedure definition. When an outside break point is reached while tracing a file, dbps stops sending code from the file and updates the context displays. Control is then returned to the user. When stepping through the file, the outside break point has no effects. An inside break point has the same effect for both tracing and stepping through the program. When an inside break point is reached, the user is presented with a dialogue box. The message in the dialogue box depends on whether the break is inside a string or a procedure definition. Break points inside strings do not make sense and the message states the break will be ignored. On the other hand, a break inside a procedure definition will present the user with a message stating a break will occur at this point when the procedure is executed (how this works is discussed later). In either case, the user must click on the *Continue* button inside the dialogue box to proceed.

Break points defined inside of procedure definitions insert PostScript code into the definition to signal a break to dbps. The following is the code that is added:

```
``line #'' /dbBreak $db

where dbBreak is defined as:
/dbBreak {
      %send dbps the break signal....
      /dbBREAK_START $db = flush
      %send the break ID to dbps,
      %should be on TOS before dbBreak called ...
      = flush
      %run the Break Executive.....
      /dbBreakExecutive $db
} bind def

dbBreakExecutive is defined as:
/dbBreakExecutive {
```

```
   { flush
      {disableinterrupts (%statementedit) (r) file enableinterrupts}
      stopped
         {pop pop $error /newerror false put enableinterrupts}
         {dup status not
            {pop (quit\n) print flush exit} if
            {cvx exec}
            stopped
               {$error /newerror get
                  {/dbERROR_START $db
                   errordict /handleerror get exec
                   (\n) = /dbERROR_END $db /dbEND $db}
                  {exit} ifelse
               } if
         } ifelse
  } loop
} bind def
```

When a procedure is executed which contains a break point, execution will stop when the break point is reached. The context displays are updated and the user is presented with a dialogue box. The message in the dialogue states that a break point has been reached inside a procedure at a particular line number. The dialogue also contains two buttons: *Continue* and *Delete Break*. *Continue* will continue the execution of the procedure and *Delete Break* will remove the break point. Removal of the break means that when that point is reached again, execution will not stop.

While the dialogue box is up, the controls on the file debug window are turned off but the user has control to manipulate the other windows. This insures that the execution of the procedure will continue from the point at which it was stopped. Before clicking on the *Continue* button, the user can manipulate the other windows as he/she could any other time. Care must be taken on the user's part not to change the state of the interpreter in such a way as to cause the program to get an execution error.

5.8 Dictionary Browser

The Dictionary Browser in dbps is also implemented as a separate window from the main window. It is brought up by clicking on the *Dict Browse* button. This window is

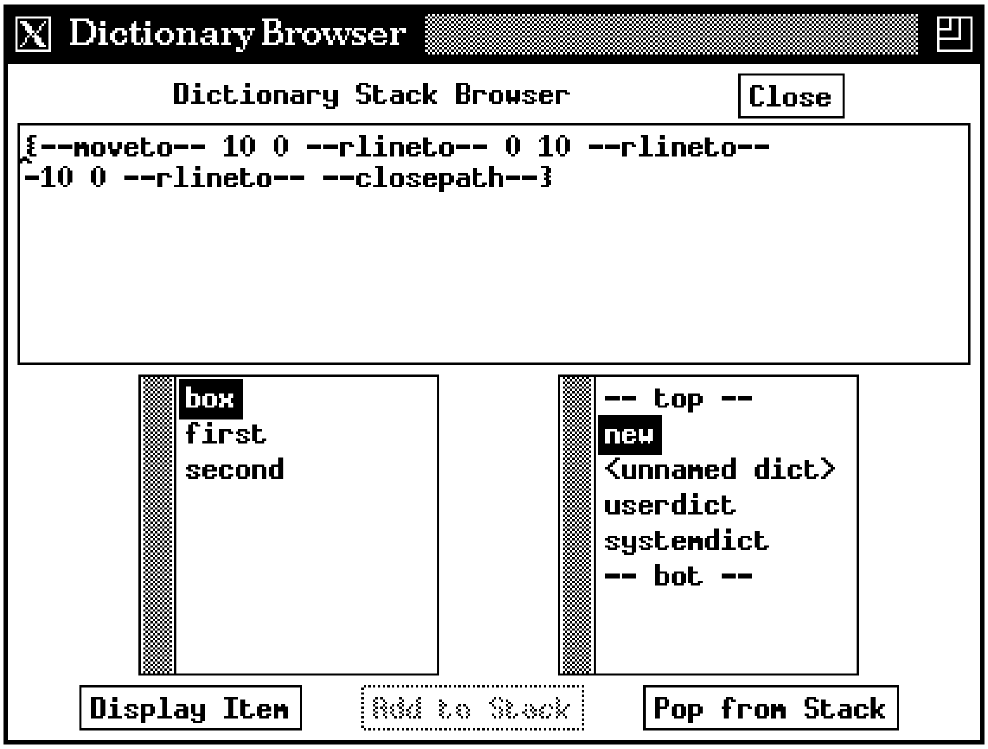

Figure 5: Dictionary Browser Window

made up of two single column list widgets and a text widget along with four command buttons as shown in Figure 5. The features available to the user are:

- List the keys of any dictionary on the dictionary stack.
- Display the value associated with a key in the key list.
- Push dictionaries on and pop them off the dictionary stack.

The lower right subwindow is a copy of the dictionary stack from the main window. Any updates to the dictionary stack are reflected in both copies. The contents of a particular dictionary are displayed by clicking on the desired dictionary name. The name is highlighted and the keys defined in the selected dictionary are then displayed in the lower left subwindow. This is an alphabetically-ordered list of keys of the key-value pairs defined in the selected dictionary. If any of these keys denotes another dictionary, an asterisk is appended to its name in the list.

Selecting a key from the key list highlights the key along with the *Display Item* button in the lower left corner of the window. The value of the selected key is displayed by clicking on this button. The value can also be displayed by double clicking on the key's name itself. The value associated with the key is displayed in the text widget. If the definition does not fit in the text widget, the entire window can be resized to increase the size of the text widget's subwindow. dbps instructs the interpreter to use the `==` operator to print the key's value to standard output. The `==` operator is used because it produces a text representation that resembles the PostScript syntax that created it. If the key's value does not have a printable representation, the `==` operator returns the name of its type in a form such as "-dicttype-" or "-marktype-".

Dictionaries can also be pushed and popped from the dictionary stack in the Dictionary Browser window. Where a name in the key list has an asterisk at the end, its value is a dictionary. Selection of this key will not only highlight the *Display Item* button but also the *Add to Stack* button. The *Add to Stack* button will push the selected dictionary onto the dictionary stack. This dictionary can then be selected to display a list of its defined keys.

Pushing a dictionary onto the dictionary stack also highlights the *Pop from Stack* button if it is not already highlighted. The *Pop from Stack* button is highlighted whenever there are other dictionaries on the stack besides the `userdict` and `systemdict` (these two dictionaries cannot be removed from the dictionary stack). Clicking on this button will remove the top dictionary from the dictionary stack. If the contents of the top dictionary are currently displayed in the key list, this list is cleared when the dictionary is popped off.

6 Conclusion

dbps demonstrates the feasibility of an easy to use and informative user interface to a PostScript interpreter. It provides the user with a more effective way to experiment with, debug and create PostScript programs. Its features allow the user to continuously observe the state of the interpreter, control the execution of a PostScript program, and interact with the interpreter during the execution of a program. The fact that PostScript is an interpreted language and that the interpreter can communicate over standard I/O channels lends itself to the addition of an interface of this sort.

The main window of dbps maintains a continuous display of the interpreter's state. The information displayed is selected by the user so only what is desired is displayed.

In conjunction with the state display the main window also contains an interactive subwindow. In this subwindow the user can interact with the interpreter before, during and after the execution of a program.

The File Debugger window of dbps provides the user with control of the execution of a program. By stepping or tracing through a program and setting break points, the user is able to manage its execution. dbps also extends LaserTalk's ability of setting break points by allowing them to be set not only outside but also inside of procedure definitions. The break points inside a procedure definition cause a break at that point in the procedure when it is executed. This assists the user in debugging complex procedure definitions.

While the features provided by dbps already greatly facilitate the user's ability to work with PostScript, there are additional features which dbps might be able to provide in the future. To make dbps more of a PostScript development environment, it would be useful to provide editing functions which are better suited for PostScript. Key bindings could be added which would assist the user with program structure and comments so PostScript conforming documents can be more easily produced. Additional features to improve dbps's file debugging facilities are another area for further investigation. It would be desirable to have features such as conditional break points and the display of user-defined variables when a break occurs.

Another area for further investigation is the updating of UCBPS from the X10 window system to using the X Toolkit under the X11 window system. The modifications to UCBPS would need to enable it to manage its own window while also being able to communicate over standard I/O channels. dbps would then no longer have to fork off a child process to monitor for events in UCBPS's window. Work in this area could also lead to making dbps general enough to be compatible with any PostScript interpreter.

References

[1] Randy Adams. Lasertalk: Postscript language development environment, 1987.

[2] Leo Brodie. *Starting FORTH*. Prentice-Hall Publishing Company, Englewood Cliffs, New Jersey, 1981.

[3] John Coker. A UNIX postscript interpreter. VORTEX internal report, Computer Science Division, University of California, Berkeley, California, 1987.

[4] Digital Equipment Corporation. *Ultrix Work System Software: DECwindows Applications Guide*, 1988. Section 10.

[5] Aladdin Enterprises. Ghostscript, 1989. Documentation distributed with software.

[6] James Gosling, David S.H. Rosenthal, and Michelle Arden. *The NeWS Book*. Spring-Verlag Publishing Company, New York, New Yorks, 1989.

[7] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[8] Adobe Systems Inc. *PostScript Language Program Design*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[9] Joel McCormack, Paul Asente, and Ralph Swick. X Toolkit Intrinsics - C Language Interface. MIT X Consortium, X Version 11, Release 4.

[10] Chris D. Peterson. Athena Widget Set - C Language Interface. MIT X Consortium, X Version 11, Release 4.