1989

GSD - An interactive window oriented debugger

Gary Bricault S.

Rochester Institute of Technology
School of Computer Science and Technology

GSD
An Interactive Window-Oriented Debugger for the AT&T UNIX-PC

by
Gary S. Bricault

A thesis, submitted to
The Faculty of the School of Computer Science and Technology, in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science.

Approved by:

11/2/89
Professor James Heliotis

10/31/89
Professor Andrew Kitchen

2 Nov 89
Professor Peter Anderson

October 20, 1989

Title of Thesis: GSD - An Interactive Window Oriented Debugger

I, Gary Bricault, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Gary Bricault

November 3, 1989

This Thesis is Dedicated
To My Precious Wife Melody
And
Daughters Elizabeth and Christine

Who also have endured five years of Graduate School.

"To the glory of the Lord Most High, and the instruction of my neighbor thereby".

J.S. Bach (1685-1750)

ACKNOWLEDGMENTS

The author would like to gratefully acknowledge the contributions in time and effort made by the following people:

Thesis Committee

Dr. James Heliotis - Thesis Committee Chairman
Dr. Andrew Kitchen
Dr. Peter Anderson

A special thanks to Mike Travers who provided timely insight when it was needed into UNIX facilities and the C language.

Abstract

Each computer program, no matter how carefully designed, may contain code entry mistakes, errors in logic, and/or anomalies that can result in unexpected outcome (also known as bugs). In order to find and correct these problems, a software tool known as a debugger can be utilized by a programmer as an aid in isolating and correcting computer programs. The purpose of this thesis is to design and create such a tool for the AT&T UNIX-PC [1] that will allow the user to function interactively within a window-oriented environment. This new debugger will be referred to as GSD (Graphic Symbolic Debugger). A study of prior art has been made in order to learn various debugger implementation techniques, their advantages and shortcomings, and to gain an understanding of methods that may be utilized within the UNIX environment for such a tool to be effective.

1: UNIX is a registered trademark of AT&T Informations Systems

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS

TABLES

1. Debugging Techniques

Debugging is a complicated activity in that, initially, only a program's external symptoms are known. The programmer must construct hypotheses about the nature of the problem and develop ways to test them. A debugger is a tool that allows the programmer to observe the internal behavior of a program at varying levels of abstraction, from source level to machine language, in order to gain more information to support new hypotheses that will ultimately lead to the solution of the original problem, Cargill[1].

There exist several general classes of debugging methods; dump, trace, monitor, symbolic, and source-level interactive debugging. The following is a summary of these techniques, from simple to complex, that are in use today.

* Dump: the most primitive form of debugging. Contents of memory locations within a given address space at the time of the failure are printed . The programmer must then manually sift through the "wreckage" to ascertain the state of the machine (and hopefully the cause of the problem) at the time of the failure.

* Monitor: a simple low-level debugging system that is often built into small ROM-based operating systems, Small[2]. It usually features with breakpoints, memory dumps, memory modifications, and in newer versions, on-line assemblers/disassemblers.

* Trace: a display of the dynamic activity of some aspect of the program, Johnson[3]. This can be a side effect of normal program execution or print statements that are deliberately placed within a program to indicate the program state after some event(s). This

information may be used to isolate a fault (E.g. when a particular variable exceeds a range).

* Symbolic Debugging: the first example of "High-Level" debugging that allows the user to think in terms of source-level names and constructs . Typically, source-level names are cross-referenced between the executable code and the source document through use of a symbol table that is created by the compiler or assembler. For example, at some specified breakpoint the debugger can reference a program's memory address back to the symbol table to show the current value of a variable in the source document. Some symbolic debuggers cannot contend with indirection and therefore cannot display a structured item, Johnson and Kenney[4].

* Source-Level Interactive Debuggers: a high-level debugging technique in which knowledge of the underlying architecture (e.g. registers) of the machine is no longer needed . All debugging is carried out against the source document itself. Program references and structures are directly related by either line number and/or symbolic name to the source document. The command-line interaction is replaced with a multiple-window bit-mapped CRT display environment, Johnson[3], Johnson and Kenny[4], Cargill[5]. Included on the display may be the source document, variables and structures, program back-trace, pop-up menus, etc. User interface is provided through both mouse and keyboard command entry. One clever approach that has been devised is to draw data structure diagrams automatically, Mateli and Radack[6]. The purpose is to create a graphic representation of a structure based upon information that may be obtained from the variable declarations and the current

values of its members. Yet another approach is to replace static information displays with dynamic displays (animated) to show the state of structures as they are updated, Moran[7].

Within the existing UNIX-PC are two options for debugging tools; ADB [1] and SDB [2]. ADB is a general purpose command-line oriented debugger that features attributes of the monitor debuggers. Its primary use is to examine and modify memory or files and also run UNIX programs. SDB, a symbol debugger, is an expanded version of ADB for use with high level languages such as "C". SDB is command-line oriented and can refer back to the source document as in a source-level debugger. Both of these debuggers require a memory dump 'core' file [3] and the target object file [4] that has been compiled to include additional debugging information [5]. This compile time information consists of source reference line numbers and additional symbol table information that will enable SDB to relate absolute addresses to source line numbers or procedure labels, perform stack traces with symbolic output, and many other useful operations.

1: ADB(1), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T
2: SDB(1), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T
3: CORE(4), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T
4: A.OUT(4), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T
5: CC(1), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T

2. Comparison of Available Debuggers

As stated previously, the UNIX-PC is supplied with the command-line oriented debugger, SDB. A literature search has found three commercially available window-oriented debuggers that have features that are desirable to include; Dbxtool, Adams and Munchnick[8], developed for Sun[1] Workstations , MacMeth, Wirth et al[9], and DMDPI[2]. Before it is possible to design a window-oriented interface to SDB, it is first necessary to examine the features of the four named debuggers, determine which features of SDB can be utilized in the windowing environment, and finally to select those features that will be possible to implement in the UNIX-PC environment.

The following is a summary of features found in SDB and the commercially available window-oriented debuggers.

2.1 SDB

SDB is a command-line oriented symbolic debugger that is used with C programs. It has the capabilities to:

* Print a stack trace.
* Print the value or address of a variable in a multitude of different output formats.
* Modify the value of a variable.
* Examine source file documents, switch between files or

1: Sun is a registered trademark of Sun Microsystems, Mountain View, California, USA
2: DMDPI, in "AT&T 630 Terminal Software User's Guide", AT&T

directories, and search through documents using a limited form of regular expressions as found in ed [1].

* Print the value of machine registers and the current machine language instruction.
* Print current machine language instruction(s).
* Run controlled execution of a program. This includes setting and deleting breakpoints in a source file, single stepping through source or machine language code line by line or to the nth step, or until a variable or address has been reached n-times or a variable has been modified.
* Execute a procedure with specified values for the argument list.

SDB is a very complete debugger in that it has capabilities that allow the user to do practically everything imaginable in examining, modifying, and running a program in a command-line environment. Its primary drawback is that, in its flexibility, the command structure can be very confusing to operate. Also, as each debugger command is executed, it produces only one piece of information, usually in a line by line output that gets lost as it scrolls off of the screen. This makes it difficult to construct the "bigger picture" of where the problem in a program might be and further reinforces the need to place the maximum amount of useful information into multiple windows where it is readily available to the user at all times.

1: ED(1), in "AT&T UNIX System V User's Guide", Vol. II, (1985-1986), AT&T

2.2 Dbxtool

As a window- and mouse-based interactive debugger for C, Pascal, and FORTRAN, Dbxtool is designed to increase programmer productivity by extending the facilities of the dbx [1] debugger. The following is a brief list of Dbxtool features;

* The environment consists of five sub-windows: a status window, a source window, a menu of command buttons, a command dialog window, and a variable value display window. The status window displays the file name and line number range of the code in the source window. The source window displays the current locus of execution and can be scrolled or searched via regular expressions. The button window contains a menu of 6 commands that can be "constructed" (see below) from the mouse. The display values window contains values of selected variables whenever execution of the debugger is suspended.
* The button window was chosen over pop-up menus because the vast majority of debugging actions consist of six default commands. The default form of the window uses very little screen space and does not obstruct portions of the screen, as a pop-up menu would. Commands are "constructed" by the selection of text and clicking a command button (or entering a command from the keyboard). This specifies a built-in function that takes the selection, as its argument, and returns a string. The text portion is appended to the string and then passed as a command to the underlying debugger.

1: DBX(1), in "UNIX Programmer's Manual, 4.2 Berkeley System Distribution, Volume 1, Computer Science Div., Univ. of Ca., Berkeley, Ca. (August 1983)

* The user has the ability to control breakpoints. A source line that has a breakpoint associated with it is marked with a special icon. The conditions for a breakpoint are similar to what is found in SDB.

* Dbxtool has the capability of running either single or multiprocess programs in a controlled manner. Execution may proceed normally, to a breakpoint, in a single step mode, or be halted.

Dbxtool is implemented as two processes; a window-based front end and a slightly modified version of dbx, connected via a UNIX pipe. Dbxtool invokes dbx with a special command line flag telling it that it is running under Dbxtool. This division of labor has several advantages, the most important one being that its own maintenance is reduced as there is only one version of dbx subject to changes. The design of Dbxtool is well thought out in terms of features and user ergonomics and is clearly a good model to influence any new debugger designs.

Can dbx, and consequently, Dbxtool be ported to the UNIX-PC ? No, not easily. The Dbxtool interface is dependent on its CRT screen environment and would have to be rewritten to utilize the screen manager of the UNIX-PC. The author feels that such a project is too ambitious for this thesis.

2.3 MacMeth

The MacMeth debugger was developed as part of a larger software package for the MacMETH Modula-2 Language System for use on the Apple Macintosh. It is tailored for the modular nature of the Modula-2 programming language. Within the MacMETH environment the

user can edit, compile, and run Modula-2 programs . If a run time error is encountered while working within the development environment, the debugger is automatically invoked at the point of the error. Features of the MacMETH debugger are as follows:

* The debugging environment consists of five windows: a source window, a procedure chain window, a module list window, and two feature programmable windows named Data 1 and Data 2. The source window contains the program source code (if it is available) and it is initially positioned at the location of the run time error. The module window contains a list of all loaded modules. The procedure chain window indicates the reason for the program error and shows the calling sequence of the active procedures. The default Data 1 window contains the variables of the last called procedure and the default Data 2 window contains the variables of the module containing that procedure.

* All windows may be scrolled, relocated, or zoomed in a manner that is consistent with standard Macintosh software. Global commands are accessible through pull-down menus at the top of the screen or keyboard entry.

* Data variables in the Data 1 and Data 2 window show the type and current value. If a display line shows a value of '*', it denotes that the data type is complex (e.g. a record, array, etc.) and further information may be obtained by clicking that line. These structures or arrays may then be exploded and examined element by element. For example, a linked list could be manually traversed and analyzed. In addition, the top line of the window contains a display of the path of pointers through which the present structure was

reached. This path can be retraced in reverse order by selecting a previous component.

* By clicking a line in either the procedure chain or module list window, the source window and data windows can be immediately relocated to any active point in the running program.

The author has used the MacMETH debugger in the Modula-2 environment and found it to be a very effective tool. The ability to view simultaneously multiple pieces of pertinent debugging data as well as access and follow data structure members represent a significant improvement in debugger design and user productivity.

One shortcoming is that the debugger does not provide a means to set breakpoints or to run a program in a controlled fashion. Since breakpoint usage is not provided it is necessary to insert a HALT instruction at the desired breakpoint in the source code, recompile and rerun the program. When the HALT instruction is encountered the program execution is suspended and the debugger is invoked.

Could the MacMeth debugger be ported to the UNIX-PC ? No. This debugger is specifically designed for the MacMeth Modula-2 development environment on a Apple Macintosh. There is no common point in either the physical architecture of the Modula-2 language or object file structure that will allow a reasonably simple conversion to be made.

2.4 DMDPI

DMDPI is a window-oriented debugger in much the same sense as Dbxtool or MacMeth. However, its operating environment is quite different in that the debugger does not run within the host

computer. Instead, the debugger application is downloaded into an intelligent terminal where it controls the user's windows and monitors the program's operation. This is very similar to the Cargill[5] Blit (Bit-Blt bit mapped) Virtual Terminal running joff, a debugger for C programs running on the Blit itself. Thus, the terminal can be tailored to run any application by simply downloading a graphics based program to it from the host system.

The DMDPI debugger is run on an AT&T 630 Terminal. This terminal has a large very high resolution CRT screen with its own 68000 based on-board system. The DMDPI must be run on the terminal under "layers" in order to initiate the debugging session. Layers allows a number of UNIX system shells to be run in separate rectangular areas of the screen, that in essence, behaves like separate ASCII terminals. Each shell region operates asynchronously thus giving the user the ability to create, move, reshape, delete, and overlap many layers with a graphic mouse. The mouse controls the way that the layers overlap, and selects the active layer to which keyboard input may be directed. Any layer that is overlapped or obscured remains active and may be written to at any time. Its contents is visually restored when the layers are rearranged. By giving the user the ability to function in this manner, multiprocess programs may be debugged by initiating a separate debugger process for each of the target processes that it runs.

The DMDPI debugger is similar in many ways to the other debuggers sited earlier. Only some of the major differences will be outlined.

* Window Organization: Because of "layers", DMDPI runs on a very high resolution CRT screen with each instance of the debugging

process functioning as its own virtual terminal. Its windows are not initialized to a specific fixed size or located in any particular area on the screen as in the cases of Dbxtool or MacMeth (even though these windows may be moved or reshaped). Upon creating an instance of a DMDPI debugger process, the user defines the debugging environment window by "dragging" the mouse icon to form the border outline of the desired size. From then on, all of the activities for that instance of the debugger is confined to the boundaries of that window, including information windows and command menus.

* Command Menus: Dbxtool and MacMeth have mouse activated command menus located in fixed areas of the screen. Dbxtool uses a center screen button bar to build instructions, while in contrast MacMeth uses pull-down menus located at the top of the screen in typical Macintosh style. Because of the random location and sizes of the DMDPI windows, variable size pop-up menus provide the only logical means for command selection. When a mouse operation requests a command menu, it appears at the location of the mouse icon within its window and contains a item list that is appropriate for the desired window. It is present as long as the mouse button is held down. Because of the need to restrict all activities of an instance of a debugger process within the boundaries of its window, a command menu with many selection items may not be able to fit within its process window. The solution is to size the menu as required and scroll the selection items up or down by pressing the mouse icon against the upper or lower boarders of the menu window. Furthermore, if a selection item results in the creation of more decision levels (limb of a tree), its item line will contain an "->" arrow. If the mouse icon is pressed to the right of the arrow, a new pop-up

sub-menu will appear with the new level selection list. Thus, command strings can be built by following a decision tree with the mouse icon. When the mouse button is released, all of the selected element(s) of the pop-up menu(s) are combined and executed. If a command confirmation is required, the process window will indicate its request by changing to a lower density. If user keyboard input is required, a prompt for input is issued.

* Window Updating: Because each information window of DMDPI is running as a separate process, it may be written to even if it is partially or fully obscured.

* Display Variables: The display of the values of variables, if requested, is divided between two windows, one for global and one for local variables.

* Breakpoints: Breakpoints may be set or cleared at any place in the source code window. A source line in which a breakpoint is set is highlighted in reverse video. A list of all active breakpoints can be requested to be displayed in a sub-window.

* Controlled Runtime Environment: Target program execution can be controlled in a variety of modes including run to a line breakpoint, single stepping, or run until a data variable condition is met. Any time that a program is halted, all windows that are associated with that process are updated to the status at the time of the halt. In addition, a "running" window can be requested that displays the process information associated with the target program.

DMDPI is a well designed debugger with every imaginable feature present to aid in debugging both single and multiprocess target programs. It provides tremendous flexibility by allowing the user to

request only the information sub-windows necessary for the debugging task at any given time. The pop-up menus provide a convenient, quick command entry device that guides the user through multilevel decisions and is present only as needed to construct the command string. This prevents dedicating valuable screen resources to a menu area.

The author has only two objections to the operating features of this debugger. The first, is that the item selection within a pop-up menu is made on release of the mouse button. If the mouse button is released prematurely, an undesirable command might be issued. Secondly, it is possible to create so many windows on the screen at the same time, that it becomes very easy to confuse one window's operations with another.

3. The UNIX-PC Environment

Dbxtool, MacMeth, and DMDPI function within different operating environments. Before the GSD debugger is presented, a brief overview of its operating environment, the AT&T 7300 (3B1) UNIX-PC , will be made.

* The architecture of the basic UNIX-PC is built around a 10 MHz Motorola 68010 CPU with a minimum 1/2 Mbyte of memory (expandable to 4 Mbytes) and a virtual address space of 4 Mbytes. Mass data storage is provided by a fixed 10-67 Mbyte hard disk and a removable media 370 Kbyte 5 1/4" floppy disk. The CRT display is a monochrome 720 by 348 pixel (28 lines by 80 character) bit-mapped graphics monitor for information display. User input is achieved through a mouse and keyboard. The computer may connect to a telephone line via a built in Bell 103/212 compatible modem. Additional I/O devices may also be connected through a parallel printer port and a DTE RS-232-C serial port. A built in expansion bus allows option boards to further expand memory, serial ports, and other useful features.

* The existing UNIX System V debuggers are command-line driven. Even though they get the job done, the user interface may be compared with working in a line editor vs a screen editor. A resource, such as a debugger, can be enhanced through use of the Terminal Access Method (TAM)[1] graphics package to provide multiple windows for various display operations and the user interface, similar to the windowing debuggers described earlier.

1: TAM(3), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T

* Debugging line number and symbol table information is provided through the compiler when the -g option is invoked. This information is well documented as the Common Object File Format (COFF)[1] and is necessary in the development of a source-level debugger.

* When a program is terminated abnormally a core file is produced. It contains user and system information at the time the program was halted. Through this file it is possible to reference values to symbols, perform stack traces, etc. Because the core file is totally system dependent and may differ dramatically between operating environments, there is no standardization as exists in the COFF format file. However, there are several system header files that describe portions of the overall makeup of the core file.

1: Common Object File Format, in "AT&T UNIX System V Programmer's Guide", (1985-1986), Chap 18, pg 1-67, AT&T

4. The GSD Debugger for the UNIX-PC

Having examined different debugging techniques, three commercially available debuggers that are currently implemented on other computer systems, and the environment of the proposed target system, it is now appropriate to outline the features incorporated in the design of the GSD debugger.

The development of the debugger followed a two-tiered design approach. The first phase was the implementation of SDB on the UNIX-PC by porting it from the source code of the 3B2 system. The second phase was the design and integration of the windowed-oriented environment for controlling the debugger and viewing the results.

In developing the windowing environment there were several issues that had to be considered. These were;

* Should the overall debugger be implemented as a single process or should the debugger, and each window controller, be a separate process? Since one of the goals is to present information in multiple windows that relate to each other, each window must be kept current to whatever change(s) were made within any other window. In a single process debugger, it is easier to share global information in the debugger with all of windows. However, in order to update windows that are covered or in background, it is necessary to bring any window to foreground (explicitly selected) before it can be written to. This can result in a windows flashing on and off creating an annoying effect. If the multiprocess approach is taken, windows can be updated without being selected to the foreground condition. The problems associated with the multiple process approach is that the window processes cannot directly access the global information available within the debugger as well as the

debugger not being able to directly update the window when necessary. The solution was to have the debugger and window processes communicate through interprocess queues. They not only provide for information exchange but also allow the processes to synchronize with each other by selective blocking on a queue until specific information types are present. The multiple process approach was the choice for GSD.

* What features of SDB can be implemented in a windowing environment? Because SDB is a command line oriented debugger, certain aspects of it do not lend itself well to a multi-window environment. For example, when SDB outputs information to the user, its output is created through a series of print statements that are spread throughout the code. In order to collect and format the data properly, it is necessary to track each output thread and buffer it so that the information may be output as either a single print statement if in the command-line mode, or to a window queue if in the windowing mode. Other issues such as passing error messages to the correct window or synchronizing input requests to data output actions are also dealt with in a similar manner. To provide for this information interchange, a special interface had to be created to act as a gateway between SDB and the window processes.

* What limitations to a multi-window environment does the UNIX-PC impose? The UNIX kernel allows for a maximum of ten windows at any time. The debugger requires at minimum the use of four windows. If the system window allocation is exceeded, the debugger cannot be allowed to start. Since the CRT screen of the UNIX-PC is limited in size, it is necessary to restrict the number of debugger windows and the amount and types of information that they may contain at any

one time.

* Can the debugger be controlled in windowing mode without using a mouse as a pointer device? All commands are designed to be accessible through either the mouse or the keyboard. The only exceptions to this is the relocate and resize patches of windows 1-3 as they must be manipulated via the mouse. Also, the windows do have independent zoom capabilities. SDB style commands may be entered directly from the keyboard, but command entry in this manner may not always cause every window to be updated in the same way a function key would.

Having examined the above implementation issues, the following represents the design strategy for the multi-window debugger;

* Three bordered information windows: a source window, a multipurpose items display window, and a variables window. Each of these windows will have the appropriate TAM features, such as scrolling arrows patches, relocate and resize patches, etc. Each can be moved and resized independent of the other. A fourth full screen borderless window is available for the controlled program run mode.

* Below the windows is a general purpose status prompt line, a command entry line for manual data input, and the function key patches that are reconfigured for the different menu levels.

Figure 4.1 shows how a debugging scenario looks that incorporates some of the best features of SDB, Dbxtool, MacMeth , and the DMDPI debugger.

The following sections contain summaries of the purpose for each window.

4.1 Source Window

The Source Window is used to display a source document with line numbers relative to the beginning of the source document. This window is initially positioned at a point where the program is halted (by either a run time error or breakpoint) if the associated core file is present. Along the window's top border is the window title, the source file name, and the size of the source file. Positioning within the window is accomplished by means of either mouse or keyboard scrolling, entering a desired line number, searching on a regular expression, or entering a new file or function name. This window can also be updated through the Display Items Window as in a manner that is described later. Also in this window, line number or data breakpoints can be set, displayed, or cleared. If a line breakpoint is set, the source line is marked at the location of the actual breakpoint by highlighting it in reverse video. The Display Items window is then updated to reflect the current list of breakpoints.

4.2 Display Items Window

The Display Window is used for a variety of different status and information displays such as the machine registers, stack trace, breakpoint status, debugger memory map, process list, and a function/file list. The machine register, stack trace, and breakpoint status can be updated every time some event causes their contents in SDB to change; such as setting a breakpoint in the Source Window. The function/file list contains the names of all of the program's functions and the name of their associated source files. The function name list can be scrolled or searched by function name or used as a cross reference to the Source Window. For

example, if the mouse pointer is clicked on a line with a function name, the Source Window will be updated to the beginning of that selected function. The user can select the desired display option and the window label will indicate the currently selected mode of display.

4.3 Variables Window

The Variables Window is used to display the current values and types of global variables. If local variables are declared in a function that is "active" (currently residing on the stack), those variables are displayed. Simple variables (int, char, long) will have their value directly displayed. A '*' in place of a value denotes a complex structure (derived) variable such as an array. If the line containing the '*' is clicked by the mouse, its actual structure is expanded and its elements displayed.

* Prompt Line (line 26) is used for command prompts, general status, or error messages instead of a pop-up form window. This results in faster message responses.

* Command Line (line 27) is used to input manual data when requested by a higher level command. When the user is prompted to manually input data, a cursor will appear on this line and data may be entered typewriter style. Characters may be deleted by using the Back Space key, and the entry operation terminated by the Return or Enter key.

* Function Keys (lines 27, 28) are used to generate high level commands that are appropriate for the context of each window. Each function key may be selected by either a mouse click or by pressing one of the keys F1-F8. The meaning of each key is mapped to the key patch that is associated with the currently active window and will

change with the selection of a new window or menu level.

Also, if a command selection process is tiered, the context of the function keys is reconfigured with each successive command selection.

Figure 4.1 is an example of a program that has been halted by an error. The Source Window (the currently selected window) indicates the current source file to be "try.c" and its length. The prompt line indicates the nature of the fault and the function name, source line number, and memory address when the fault occurred.

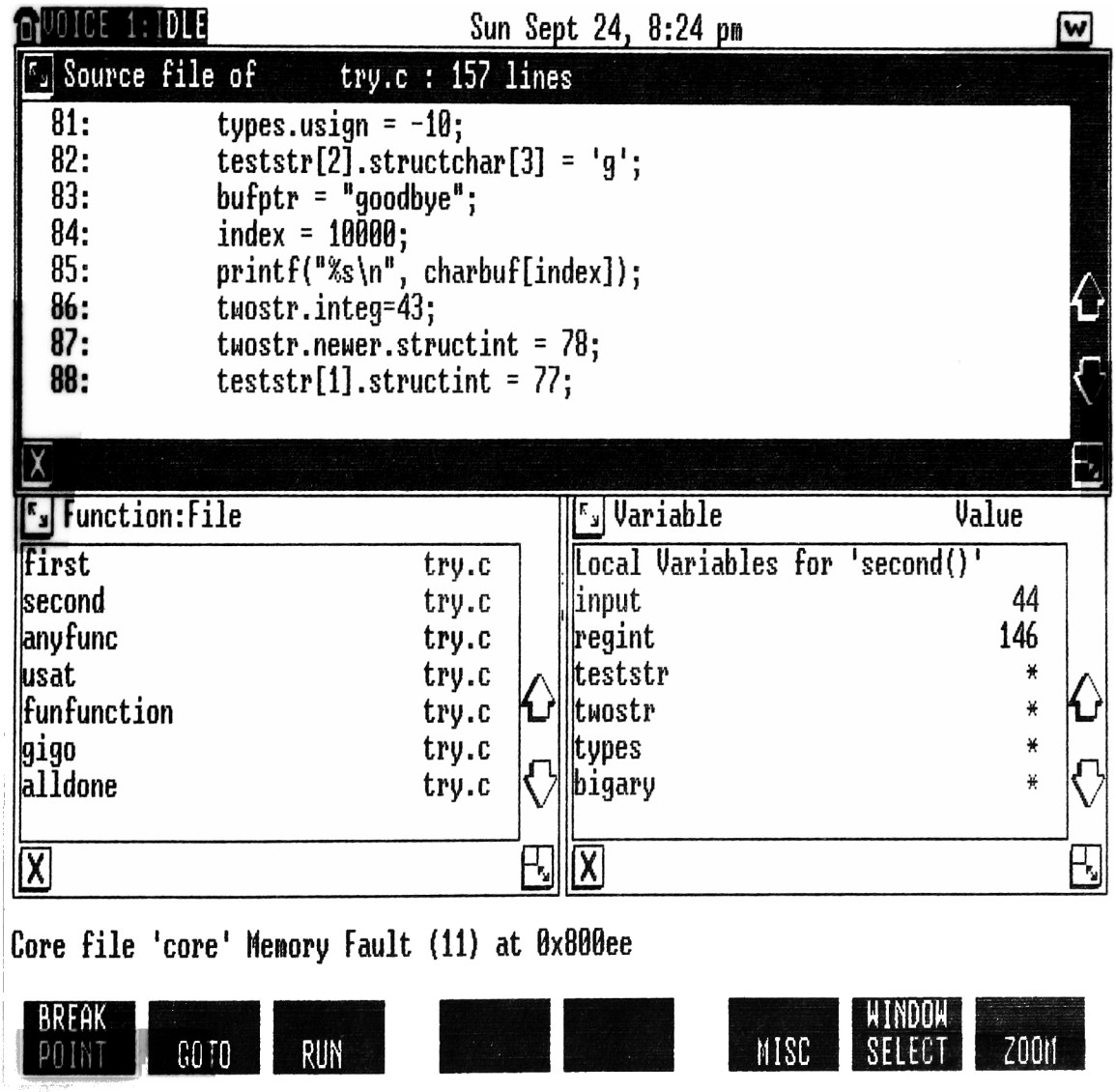

Figure 4.1 A 3 window debugging environment at the entry level.

The Display Items Window shows a list of functions in the source file set associated with the overall executable program. The Variables Window contains the list of local variables in use at the time of the program halt. The function key shows the command options available for the currently selected window (Source Window).

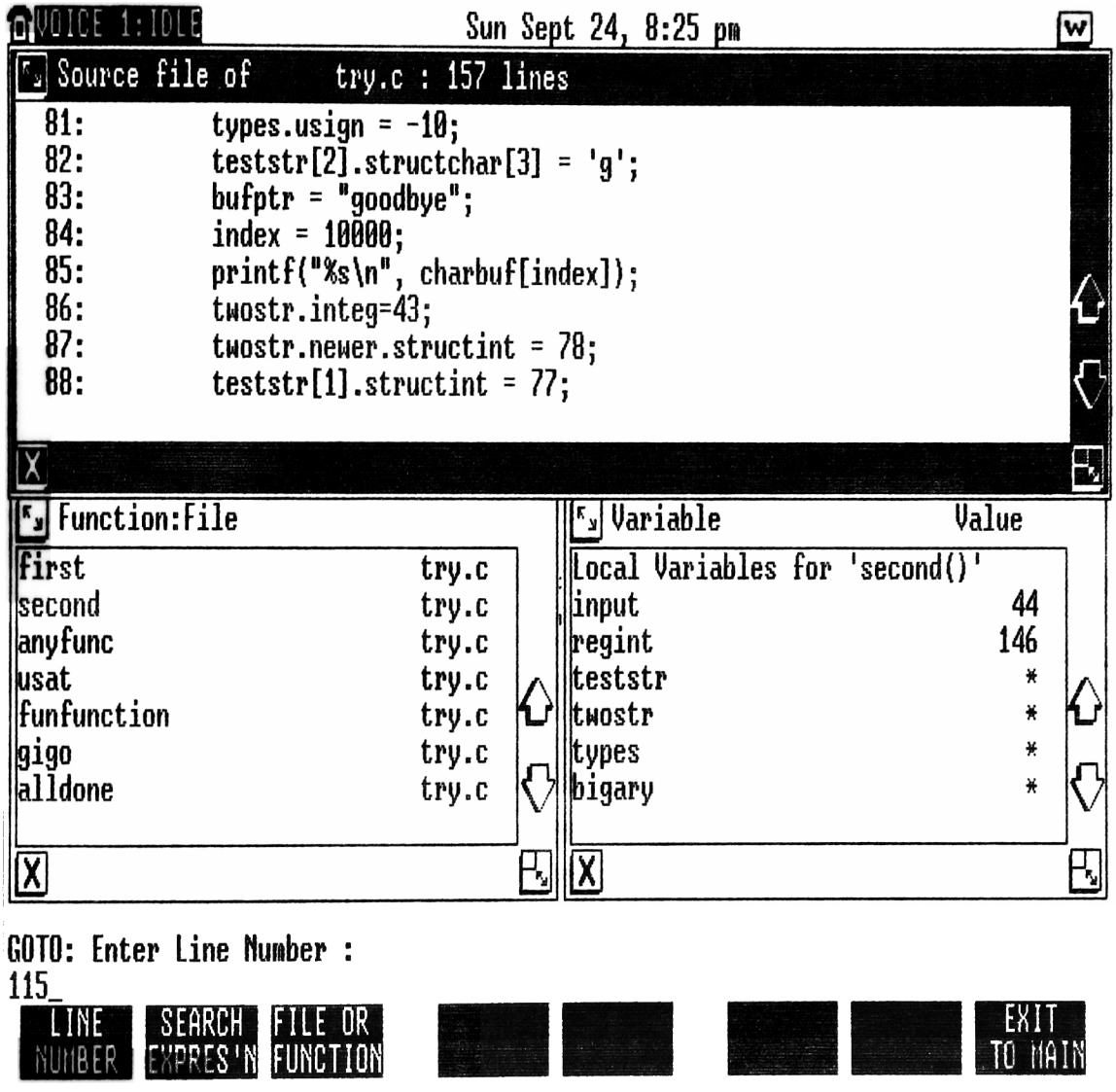

Figure 4.2 Function Keys for "GOTO" Command of Source Window.

In Figure 4.2, if the "GOTO" function key of Source Window is selected, the labeling for the function keys will be altered to

indicate the possible options, in the context of "GOTO WHAT ?", in the source window. Now the selection can be made between the options of line number, regular expression search, or a file or function name. If the Line Number options is selected the user is prompted for data input on the command line. Once the data is entered, the source window will be updated if the request can be satisfied.

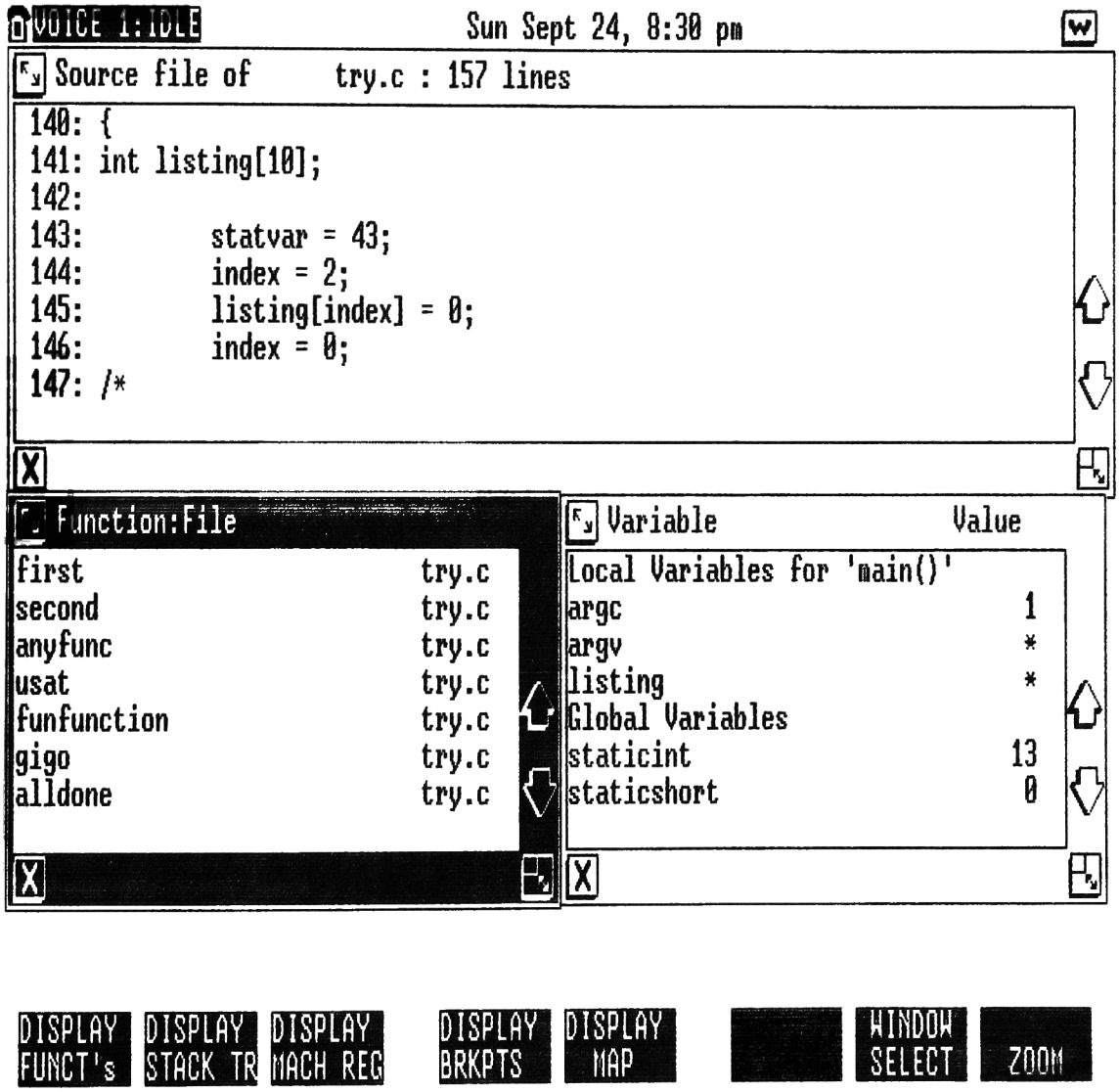

Figure 4.3 The Display Items Window is selected.

Otherwise, the "GOTO" command can be terminated by pressing

either the Exit or Return key without entering any leading characters. The function key labels will now return to their "top level" meanings.

In Figure 4.3, the function "main" is selected with the mouse. The Source Window is positioned to the start of that function and the Variables Window is automatically updated to reflect the state of the variables for that function (if the function is located on the **stack**).

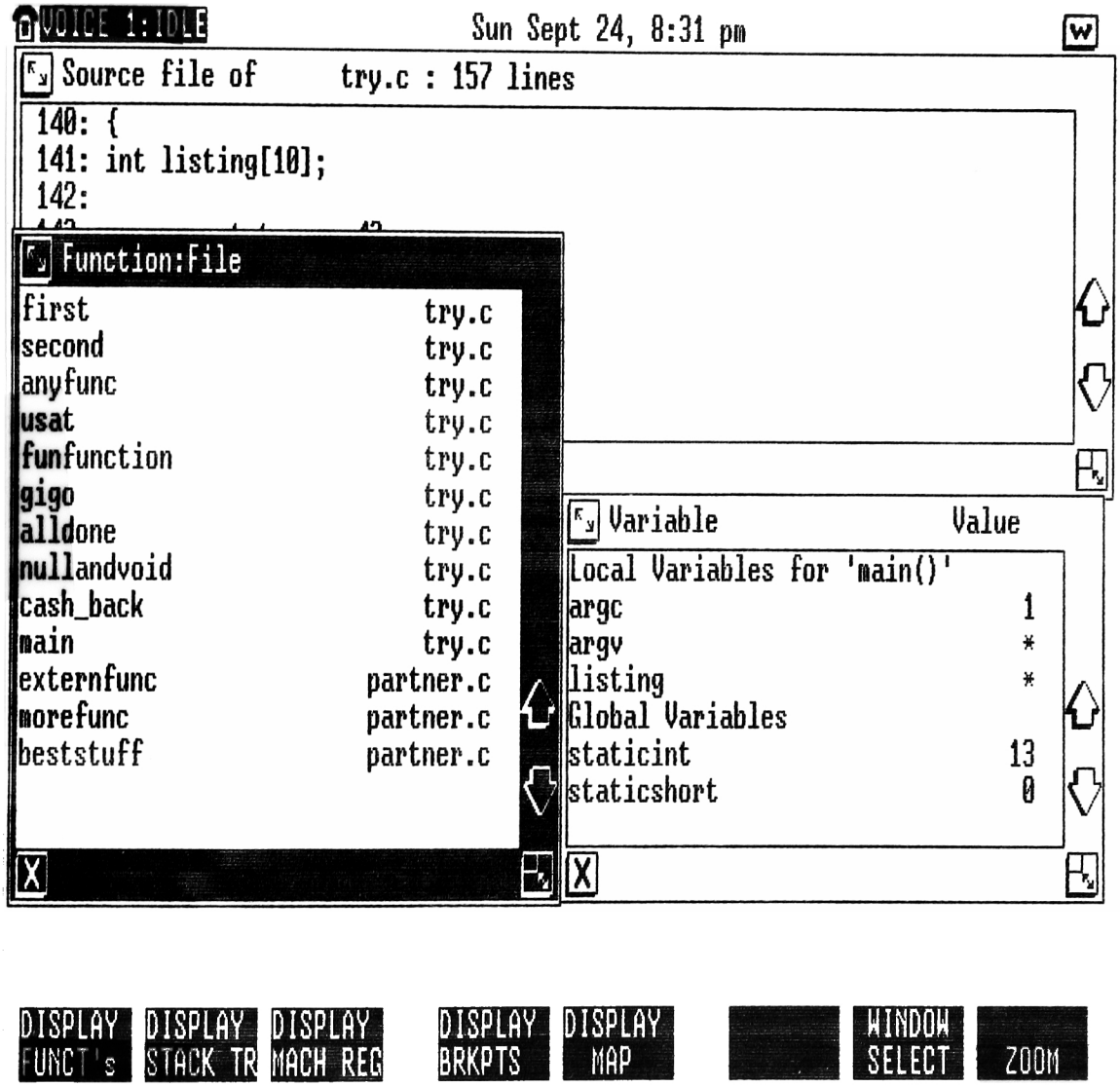

Figure 4.4 The Display Items Window "zoomed" to display more information.

This feature of the debugger insures that all windows will display current data with respect to each other. Here, the Function List in the Display Items Window has been selected showing a list of the program's functions. If more information is desired to be presented within any of the windows, the Zoom function key can be used to increase the window height as shown in Figure 4.4

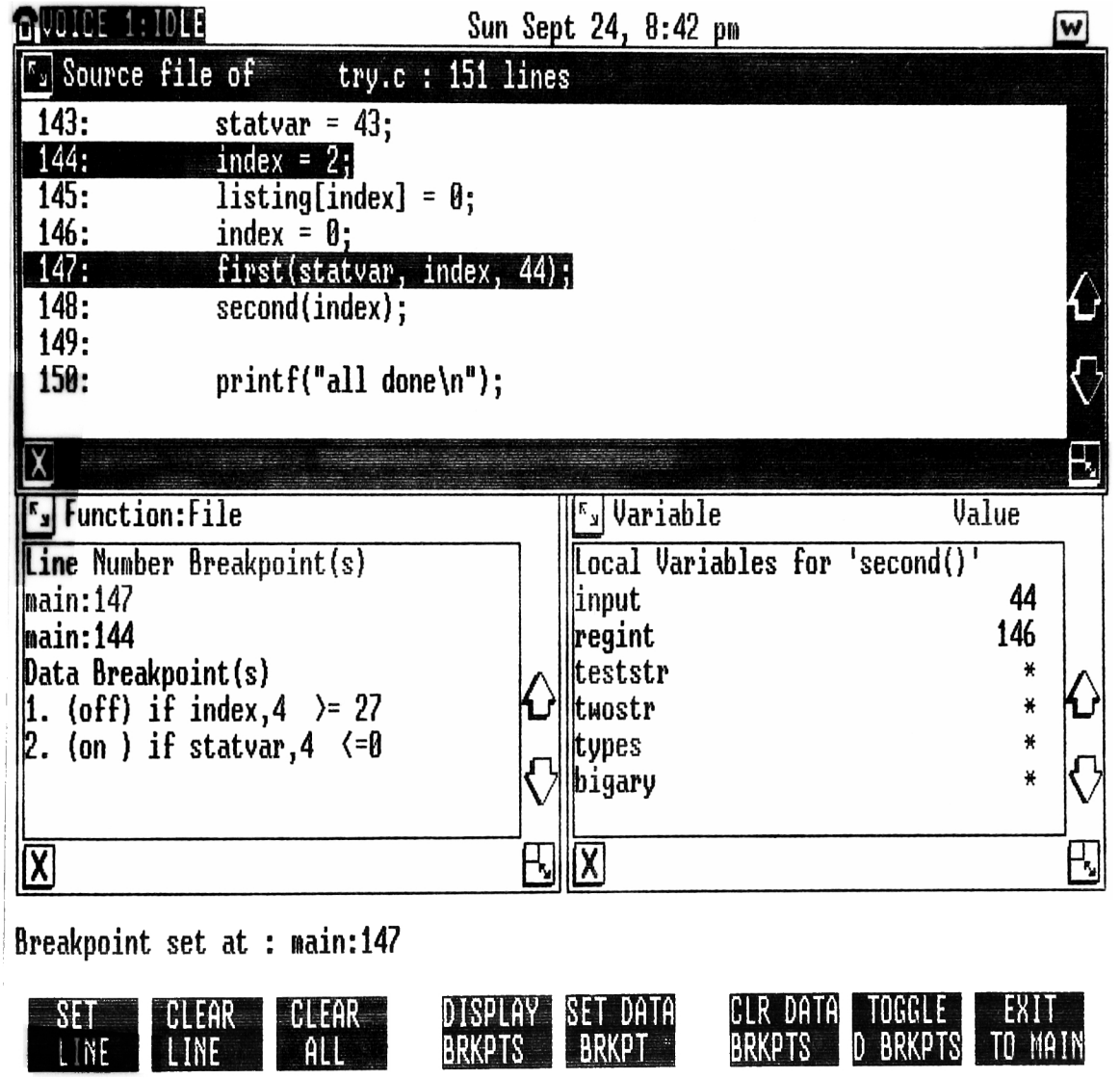

Figure 4.5 Breakpoint Operations

Figure 4.5 shows the Source Window in the Breakpoint Operations menu. From this menu, breakpoints can be manipulated for either

lines numbers of the source file (Line Number Breakpoints) or conditional breakpoints based on the values of variables (Data Breakpoints) or a combination of a line number breakpoint with a conditional qualifier.

A line number breakpoint can be set from this menu by positioning the cursor at the desired line and pressing the "Set Line" function key. In Figure 4.5, line 147 has just be set as a breakpoint. Note that source line 147 is now highlighted in reverse video, the prompt line reflects the confirmation (function:line number) of the action, and the Display Items Window has been updated with the addition to the list of previously existing breakpoints. When a line number breakpoint is cleared, the source line is no longer highlighted and the breakpoint list is updated to reflect that change.

Data breakpoints must be specified manually and do not cause any source lines to become highlighted. Also, data breakpoints may be toggled active/inactive (on/off) without being cleared. This allows the user to temporarily disable a test without having to remove and then re-specify it. Each change in status causes the breakpoint list to be automatically updated.

4.4 Runtime Window

The Runtime Window allows the user to run a program in a controlled manner. It appears as a full screen borderless window as shown in Figure 4.6. The function keys allow the user to start a program, run to a breakpoint, continue from a halted state, single step by source line or machine instruction , step until a specified variable changes, kill the runtime process or select a new window(s). An executing program,may also be halted from the keyboard

with a keyboard interrupt (^C). A miscellaneous sub-menu allows the user to select special configuration modes for output displays or other default conditions.

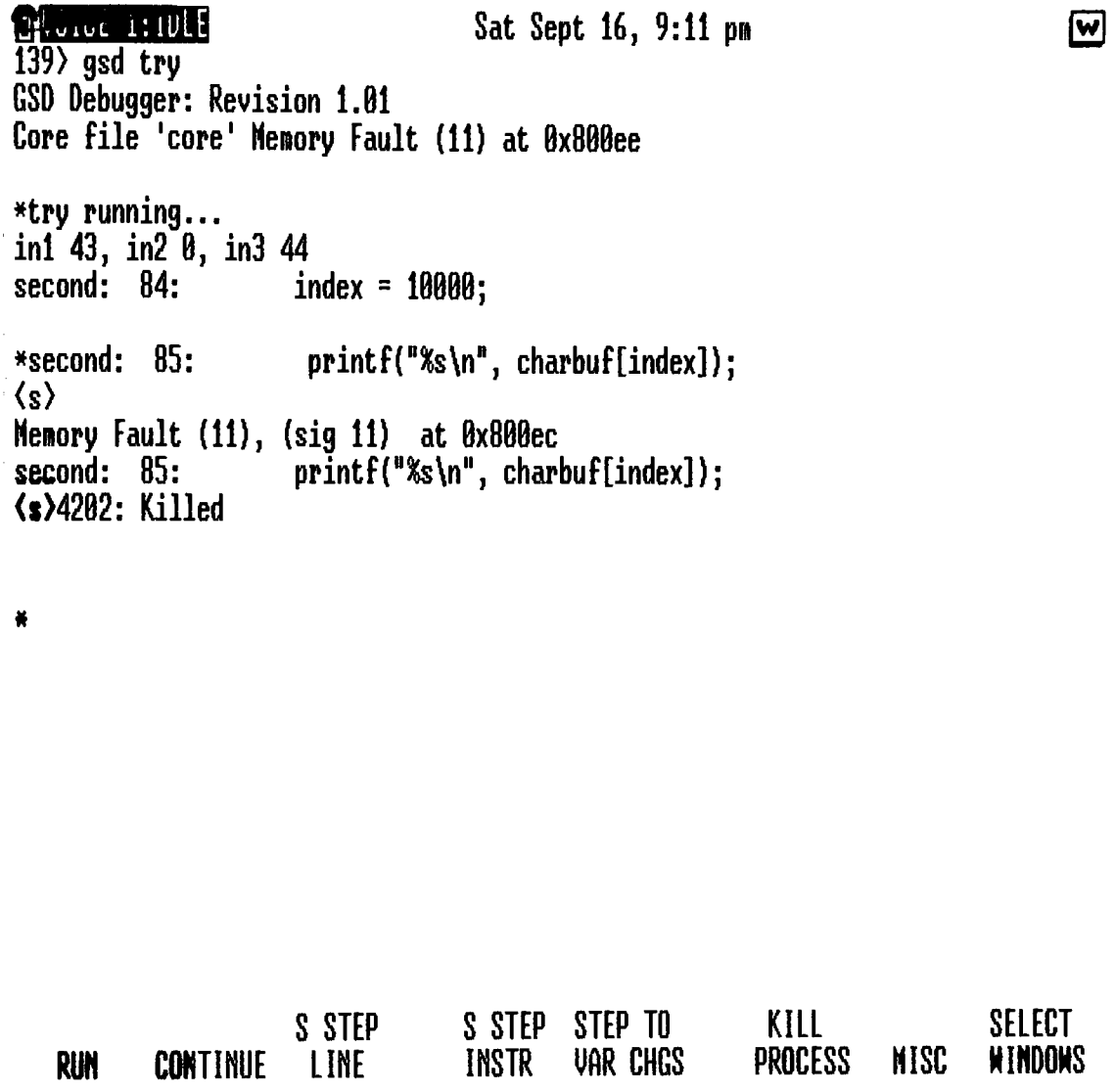

Figure 4.6 Runtime Window

In the example in Figure 4.6, the program to be debugged had generated a memory fault as shown in Figure 4.1. A breakpoint was set at line 84 and the program was RUN. Execution was suspended at the breakpoint. The program was then single stepped two lines where

5. Other Considerations

There are several other considerations in creating and using a debugger on a program at the source level: remote terminal operation, compiler optimization of code, and the potential consequences arising from use of the machines resources by the multiprocess debugger.

Using the UNIX-PC for remote debugging creates a number of special problems. The TAM graphics package allows the usage of the mouse only if it is local to the machine running the software package. If the program is running remotely, all mouse commands are ignored. Because the UNIX-PC has terminal emulation capabilities, it may be possible to operate remotely with either another UNIX-PC or a terminal, possibly over telephone lines. Command entries are via escape sequences from the keyboard. If the remote terminal supports graphics (such as a VT-100) then full windowing capabilities could be possible at greatly reduced speed with command entry restricted to keyboard input. If windowing is not supported, then remote debugging can only be accomplished by reverting to the command-line format. This, however, does not give GSD any real advantage over SDB.

Debugging optimized code raises questions as to whether or not it is practical to write a debugger that will work at the source level. Hennessy[10] states that there are many difficulties associated with attempting to debug optimized code. Only a few of these reasons are listed below.

* Loops are common targets for optimization by moving static statements outside of the loop to speed up execution.

* Most symbolic debuggers cannot report the current values of variables according to the original program. This leaves the programmer the difficult task of attempting to unravel the optimized code and determine the values that the variables should have.
* If a program is halted by either a breakpoint or a run time error, a unique machine instruction must be identified as the start of the code for a source code statement. Optimizing may affect this by changing the location of that unique instruction.
* The abilities of some debuggers to change a variable's value in an optimized program causes insurmountable problems because of common subexpression optimization.

One possible "apparent" solution to the optimization problem is to "not allow the code to be optimized before it is debugged". This approach has a drawback in that the problem may only appear in the optimized version, possibly due to a side effect. Also, some compilers optimize as part of their normal operation and it may not be possible to disable it. Because of these problems, GSD may not work reliably on optimized code, and this must be understood from the beginning.

One final consideration is the allocation of machine resources to the debugger. This must be taken into consideration, as each UNIX-PC might be supporting several users at any given time (one on the local console and others remotely logged in). It is not desirable to have the system performance severely degraded because of poor software design. The following is a list of resources used by the GSD debugger:
* Because the underlying debugger and windows are implemented as separate processes, it is necessary to provide the interprocess

communications (IPC) through queues. The debugger and each window require that a queue is created for it at start of the debugging session. As one would expect, each queue is allocated a block of memory that is dedicated to it until the queue is removed at the termination on the debugging session.

* Each of the window processes may receive asynchronous input from either the debugger, via the IPC queue, or from the keyboard. Input from the keyboard may only be directed to a window that is currently selected (in the foreground). The currently selected window process must not block on either input source if data is not present. It must continuously poll both input sources to determine if data has arrived. This polling process will utilize a major portion of the CPU's time if not carefully implemented. Fortunately, if a window is not the currently selected one (in background) its process blocks on the queue and waits for data, as it cannot receive keyboard input. The underlying debugger process never receives keyboard input, so it always blocks on the queue. When a window changes status from foreground to background, a flag is set to allow the process to block on the IPC queue. When a window is selected to foreground, the status flag is cleared, and the polling is resumed. Therefore, all processes except the currently selected window will be blocked when there is no data in its IPC queue. This minimizes the CPU allocation for each process. There also exist other possibilities for further CPU time savings that will be researched at a later time.

6. The Design and Implementation of the GSD Debugger

As outlined in Chapter IV, the goal of this thesis is to create an interactive multi-window debugging environment interface to an existing UNIX command-line oriented symbolic debugger. This approach allows the user to have continuous access to a broader selection of relevant information than would be possible with the original debugger.

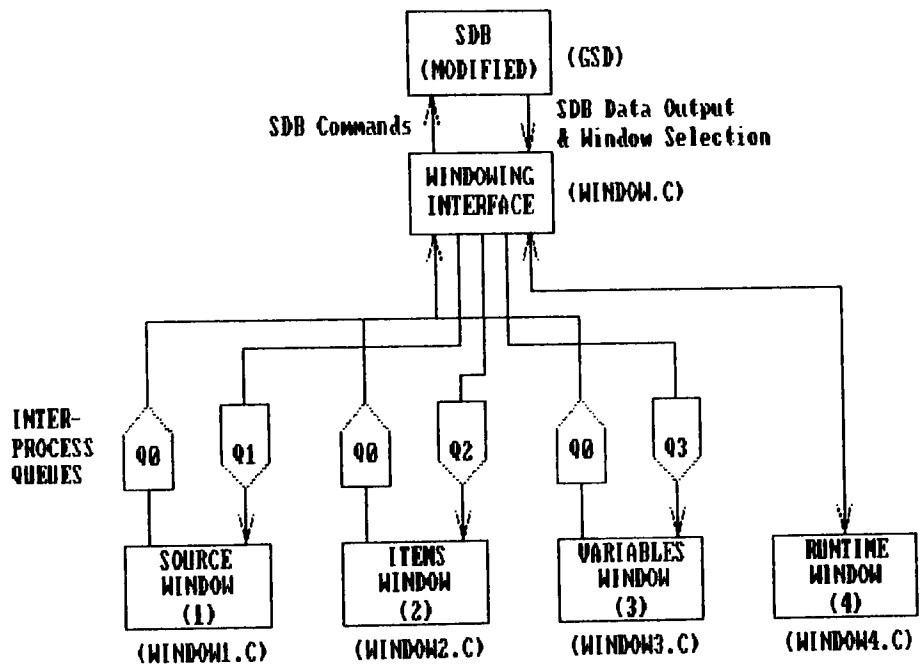

Figure 6.1 Block Diagram of the GSD Debugger

Figure 6.1 depicts the completed implementation of the GSD debugger in block diagram form. The actual debugger mechanism is a specially modified version of the symbolic debugger, SDB. In the window oriented mode, three windows known as user windows, are created as child processes that communicate with the underlying

debugger by passing messages over four IPC queues. These messages are preprocessed by a windowing interface as they are received from either the debugger or the windows. A fourth window, the runtime window, resides with the underlying debugger process. Each module of the GSD debugger will be examined in detail, both separately as a module and also how it interacts with the other modules.

A special version of SDB has been created for use by the GSD debugger. A command line flag is used to select whether the debugger will function in the traditional command-line mode or the windowing mode. The start-up sequence for the windowing mode will be explained in detail later in the text.

The design of the user interface is a series of trade-offs as how to best fit the most useful features of the debugger into a window oriented environment. This is an environment for which the debugger was not originally designed. The following is a brief summary of the major design decisions and how they were resolved:

* Which of the SDB commands should be implemented in the windows? In most cases all commands that can be described in a single keystroke (or mouse click) were implemented within the window that was most appropriate for the use of that function. Furthermore, a command can be initiated by one of several means; selecting an icon, pressing a function key or manually entering the command.

* Should the user windows be implemented as part of the debugger process or as separate child processes? The user windows are implemented as separate child processes. The chief advantage is that they can be written to whether they are in foreground or in background. This eliminates the need to explicitly select a window into foreground before writing to it.

* Will having the multiple processes polling for IPC queue or keyboard input create high system loading, thus causing poor system performance for a second user? To minimize system loading, windows that are in background block when there is no queue input. When a window is in foreground the queue and keyboard is polled in a non-blocking fashion, but at the end of each polling operation the process gives up its time slice. When the Runtime Window is in foreground the process blocks for keyboard input.

The following benchmark was created with two users logged into the UNIX-PC. During the test period user #1, on the console, was either sitting idle in the shell or running the GSD debugger, having selecting one of the four windows. User #2, at a remote terminal, repeatedly compiled a 1769 line C program to test the system load. The Table 1 lists the compilation time under each test condition.

User #1 Condition	Compilation Time(min:sec)
Sitting idle in the shell	2:06
GSD: Window 1 Selected	3:20
GSD: Window 2 Selected	3:24
GSD: Window 3 Selected	3:22
GSD: Window 4 Selected	2:11

Table 6.1 Multiple User System Loading Benchmark

* Will remote terminal debugging be possible? No. TAM does not easily support this application on remote terminals.

Each of these key points will be further explored in the discussions of the individual modules: the underlying debugger, the windowing interface, each of the child window processes and the runtime window.

6.1 SDB: The Underlying Debugger

SDB, as depicted in Figure 6.2, allows a user to perform symbolic debugging in a command-line oriented fashion by extracting information from several possible sources; the executable file (e.g. an a.out file), a core file (generated at the time of a program crash, for further information see Appendix A) and the original source document files.

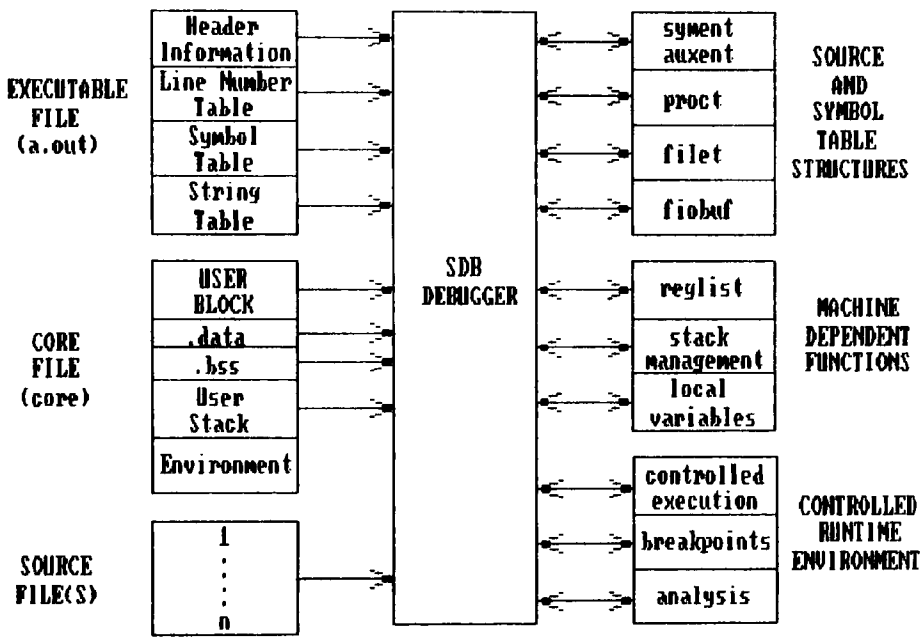

Figure 6.2 The SDB Debugger

The depth to which a program may be debugged depends on the availability of the core file and whether or not the source files were compiled to produce additional debugging information. For more information on these features, see Chapter 1. SDB reads these files and builds a series of table structures that allows the user to perform a variety of debugging functions through complex lookups and cross referencing operations.

A brief summary of the start-up sequence of SDB will help to

demonstrate how this data is organized.

* After the command line list is processed for option flags the executable file is opened and the header information is read into a map of offsets that acts as pointers to other sections of the executable file. Now the symbol table (shstrtab) and string table (strtab) are read into their respective structures.

* Next, if available, the core file is opened and another map of pointers is created to relate the offsets within the file to physical memory space. These regions are then built into structures that represent the CPU registers (reglist), procedure pointers into the symbol table (proct) that can be used to relate addresses to source code lines (lineno), start of stack pointer and pointers into the .data/.bss space. The source document, if available, can be viewed with the current line set at the point of the presumed programming error.

* Once all of the tables are set up, the tty modes are preserved, signals for certain interrupts are set up or preserved and finally, the main command input idle loop is entered.

During a normal debugging session, additional tables may be added and removed, depending on which features are being utilized. During a controlled program execution, it may be desirable, for example, to manipulate breakpoints. Space for this table will not be allocated until such time that it is actually required.

The command set that allows the user to display or modify a variable is the most complex feature to implement. To display a variable involves a complex series of lookups to first determine if it is a local or global variable. If it is a global variable, the .data or .bss sections of the core file are consulted. However, if

it is a variable local to a procedure, the stack must be searched for all instances of that variable for that given procedure. There could be multiple instances present if recursion has taken place. The lookup procedure is complicated even more if the variable is a pointer, function, or an array or combination of any of these (e.g. array of pointers to integers) as they must be dereferenced before the value can be extracted. See Appendix C for a more detailed explanation of simple and complex variable types and how they are represented in the symbol table.

A simplified diagram of the runtime scheme can be found in Figure 6.3. Controlled execution of the program being debugged is made possible through the creation of a child process that runs under the control of PTRACE[1]. The ptrace function allows a parent process to control the execution of a child process for debugging purposes. During the time that the child process is running, the parent process is suspended at a WAIT[2] node. The child process may then be run continuously or single stepped, either by individual machine instructions or by source lines. If the child enters a stopped state, such as would occur on an error or at a breakpoint the parent will be notified by a signal. The parent process will then proceed to evaluate the returned status of the child in order to determine the cause of its termination. Using ptrace, the parent can now examine the "core image" of the child for register values, variable values, stack trace, etc.

1: PTRACE(1), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T

2: WAIT(2), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T

The parent can also continue or terminate the child process through ptrace. For example, if the user wishes to single step at the source code level, SDB must single step through machine instructions one at a time checking to determine if that instruction address falls on a new source code line boundary. Since this is completely under software control, it can be very slow. The specific use of the runtime environment window package will be further discussed in Section 6.7, Window 4: The Runtime Window.

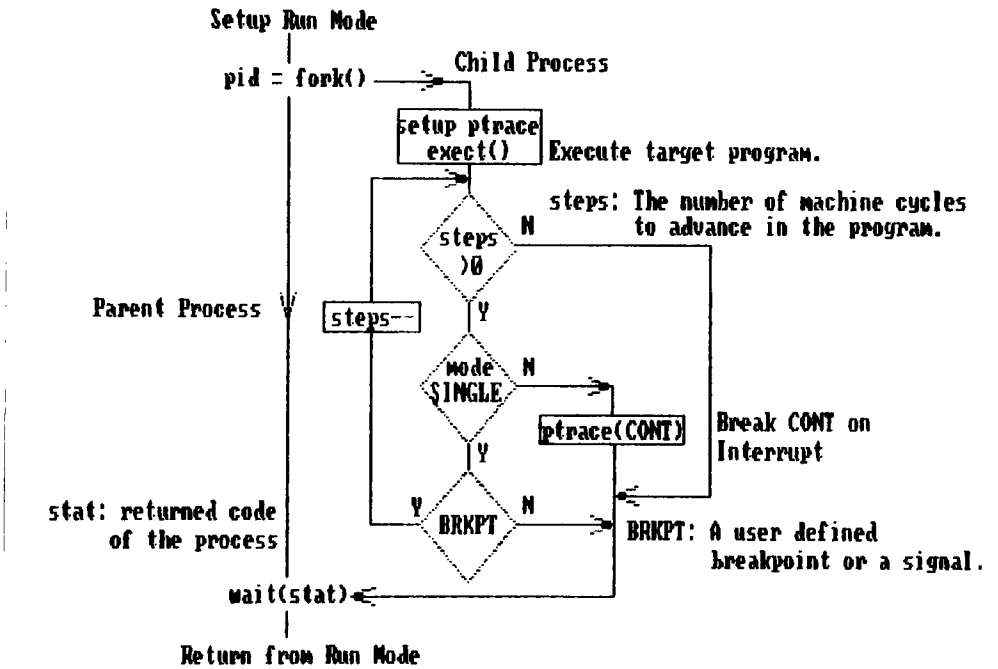

Figure 6.3 Simplified Diagram of SDB Controlled Program Execution.

There are several runtime situations that SDB has problems handling during a controlled run:

* Programs that create one or more child process(es). Because SDB itself creates a child process that in turn executes the target program, SDB cannot exercise control over any additional processes that are created beyond its own child except to break the execution with a keyboard interrupt. If a running child process is halted, SDB cannot perform any symbolic debugging, such as displaying a variable, because it does not have access to any of the "core image" of the "forked" process. Also, SDB cannot single step the target program's execution without causing an Emulation Error at the point of the program fork function call.

* Programs that can generate signals such as an alarm function. When any signal occurs during a controlled run, it will cause the termination of the child process that was running under ptrace, thus interrupting the run. The status of the child is returned to the parent. Encoded within the status is the number of the signal that caused the termination to be reported. This may be of little consolation to the user since it may not be possible to continue execution from the point from which the signal occurred.

The original SDB was designed to output its information by means of many print statements throughout the debugger. However, the new debugger must buffer the data as it follows the thread of execution for any given command. The buffered data can then be directed to the output mechanism that is appropriate for the mode of output that is selected at the time the debugger is invoked. If the debugger is set to the command line mode, the termination point of the thread will result in the buffer being printed. Otherwise, the data is written to the appropriate window.

In addition to the differences in the manner that output is

handled, user input or debugger commands may originate from a user window through IPC queue 0, from keyboard, or from function key inputs. As shown in Figure 6.4, if any of the windows are not selected (process is in background) the polling process will block on the read queue if there is currently no message present (or no message of a desired message type). When a message is received it is decoded and processed. Afterwards, the process will block again until another message is received. If a user window is selected (process in foreground) its process will not block on the message queue. Its process will then proceed to check for keyboard input as non-blocking I/O.

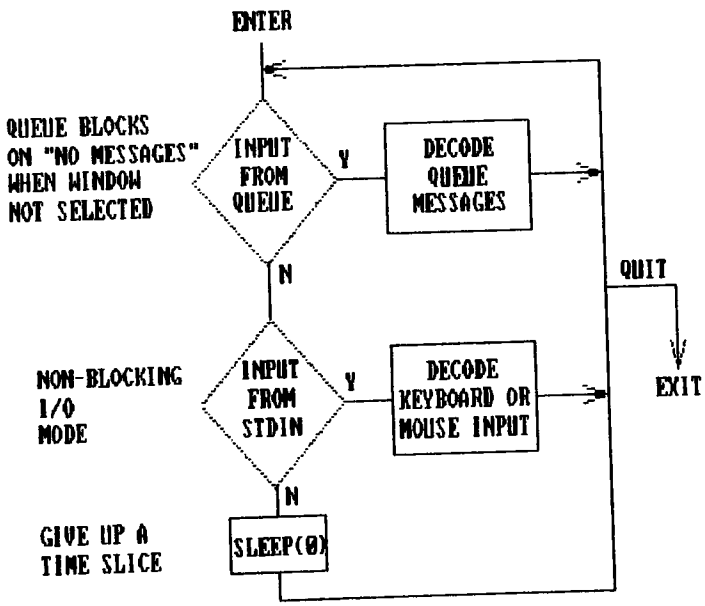

Figure 6.4 Queue and Keyboard Polling Scheme When Window is Active

However, if the runtime window is in foreground there can be no

input from another user window so the process will not check for a message in the queue but will instead block on keyboard input until such time that input is received. From whatever source, input will be decoded, processed and then the polling process repeated. If a process is blocked on either a queue or keyboard and a window status change occurs, the window signal, SIGWIND, will cause the process to go to the interrupt catcher function. There the status flags that control the blocking modes will be adjusted before the process returns to the polling mode again.

6.2 The Windowing Interface

The windowing interface block, as shown in Figure 6.1, is a special group of functions that provide a uniform interface between the GSD debugger, three user process windows and four IPC queues. One IPC queue, (0), is a common queue for transmitting messages from the user windows to GSD. The three other queues (1,2,3) are used to transmit messages from GSD to the three windows. These features are used only if GSD is set to the windowing mode. The window interface functions can be grouped as follows;

* Set up and terminate the IPC queues between GSD and the three user windows.
* Fork the child processes and initialize the underlying TAM structures and then execute the target window program.
* Broadcast the command to shut down the user and runtime windows.
* Receive and decode the queue messages that are sent by the three user windows and then build internal control or debugger commands.
* Input debugger or internal commands and use them to build commands that are sent over queues to the three windows.

* Interface and synchronize the Source Window with the source statement buffer pointer in the debugger. This will be further explained in the Section 6.4, The Source Window.
* Create program function list and send it to the Items Window. This will be further explained in Section 6.5, The Items Window.
* Create local and global variables list and send it to the Variables Window. Explode structures on command and return them to the Variables Window. This will be further explained in the Section 6.6, The Variables Window.

Each of the user windows receives its input and creates output by multiple means as shown in Figure 6.5. These windows receive input through four different means, direct keyboard input, function key input, mouse input and IPC queue input from the debugger. The method of how each window processes its input will be discussed in a later section specific to that window. Each window can generate output commands to the debugger or write formatted data from the debugger to the display for the user.

Each instance of window input is evaluated locally within that window's program. If the input is from the keyboard, function key, or mouse click, it must be first formatted to conform to the required GSD syntax and then placed on the IPC queue 0. Each queue message consists of two primary parts, the text message (.mtext) and the message type (.mtype). Because GSD receives input from the user windows over the common queue 0, it is necessary to encode the message type in such a way as to identify the window that was the origin of any given message. Otherwise, GSD would not be able to respond correctly for like message types from different windows.

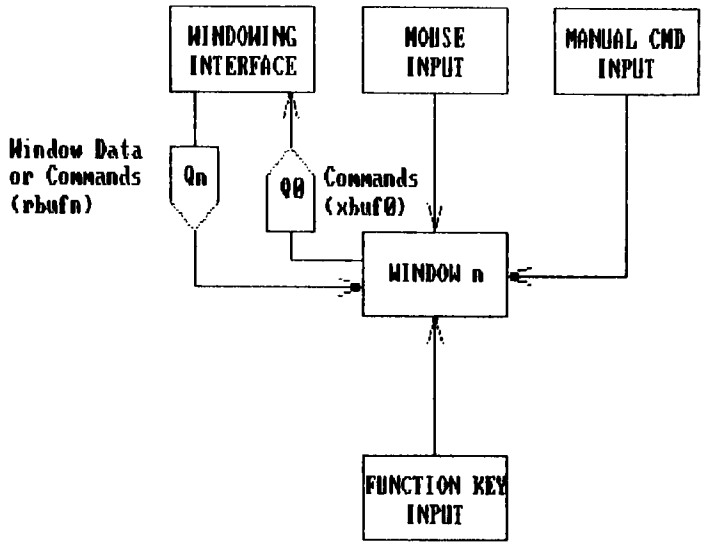

Figure 6.5 Child Process Window I/O Means

The encoding scheme that is performed in each window is as follows:

> For each Window n , there is the base message type m_id (see Appendix B), and the encoded message type mtype:
>
> mtype = (10000 * n) + m_id

Figure 6.6 is a diagram of the message decoding and processing scheme used by the receiving routines of the windowing interface. When a valid message is received from queue 0 in GSD (queue status != -1), it must first be decoded to extract the source window identifier (wid). This is done by boundary checking the message type and stripping off the identification component, leaving the base message type to be passed along with the message to the decoder. If the message type does not fall within the defined ranges, a program

error message is produced.

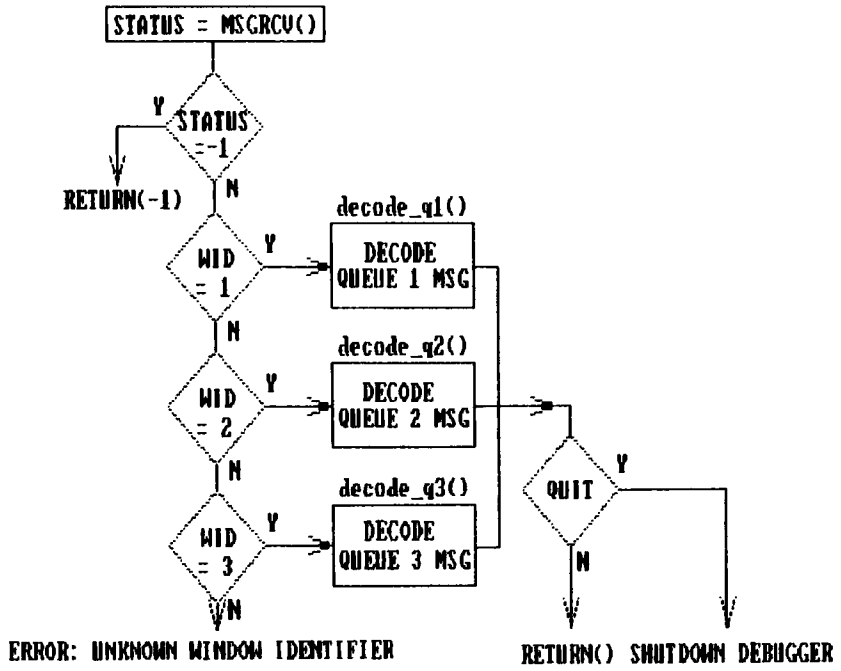

Figure 6.6 GSD Receive Queue Message Decoder

Depending on the origin window of the message and the base message type, the action that will be taken is one of two possibilities; the message type will result in one or more (a macro) GSD commands being sent to the debugger or it will create a new command to be sent back to one or more of the user windows. For example, if window 1 is active and issues a command to make window 2 active, the command is passed directly through the windowing interface to window 2 without going to the GSD debugger.

If an incoming command results in an new command or formatted data being issued to one of the user windows, the message will be sent over a queue that is dedicated to that window (e.g. window n receives messages over queue n, as shown in Figure 6.1). Therefore,

it is not necessary to encode any additional information onto the transmitted message base type.

The start-up of GSD, with inclusion of the windowing capability, becomes considerably more complex than the basic debugger. This is because the debugger design must take on the problems that are found in a multiprocess environment. In all there are four processes running, the underlying debugger and the three windows processes that it in turn initiates. Since these processes must maintain a synchronized relationship to each another, it is necessary to carefully control the start-up procedure.

Figure 6.7 depicts the start-up sequence of the debugger and its child processes. The following is a summary of that sequence:

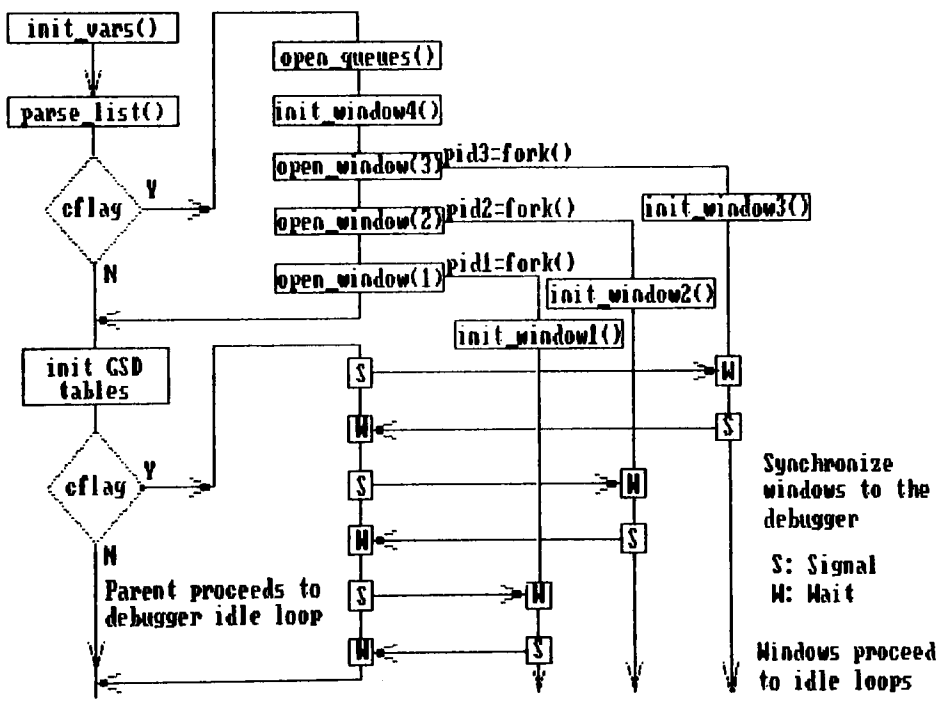

Figure 6.7 Debugger and Child Process Window Start-up and Synchronizing Procedure

* Within the GSD debugger, the main procedure, begins the start-up process in the same manner as if it were not in the windowing mode by initializing key variables and parsing the input command line for option flags. If the windowing mode is enabled, the "cflag" is set to indicate that condition.

* The IPC queues are then opened so that communication with the "soon to be created" user windows will be possible.

* Window 4, the Runtime Window, is created as device of the GSD debugger process.

* Window 1, 2 and 3 processes are created in succession. The process identifiers that are returned by the fork are used to identify the returned processes during the shutdown process that is described later. These three processes proceed to initialize their respective windows until such time that they need additional input from the underlying debugger. At that time the processes are blocked until some later time that the debugger signals it has completed its initialization procedure and that its data needed by the windows is available.

* While the child processes are initializing their respective windows, the parent process completes the initialization of the debugger tables. Then, in the order of windows 3, 2 and finally 1, the debugger signals to a window that it may proceed with its initialization. This order of initialization is necessary so that the Source Window always becomes the foreground window at the end of the start-up sequence.

* In response to the signal from the debugger, each window passes its window size to the debugger and then requests whatever other information might be necessary to complete the initialization of

that window. For example, the Source Window cannot setup its label line until the debugger has determined the point in the source code that an error had occurred (if any), the name of the source file and its size.

* Now all four processes can enter their respective idle loops and wait for input or to receive data.

In this manner, the underlying debugger and window processes can complete their initialization sequences concurrently without any danger of deadlock or phasing errors.

Because the UNIX-PC supports a maximum of ten window devices, it may not be able to start up the debugger if there are too many window devices currently in use. If it is not possible to create any one of the windows, because all of the devices have been allocated, a shutdown message will be broadcast over the IPC queues to any debugger windows that have already been created. The debugger will then close the IPC queues and exit.

Now that the design of the multiprocess start-up procedure has been discussed, it is also necessary to recognize that similar problems are also present in the shutdown of multiprocess programs. The same care must be taken to guarantee that all processes "gracefully" shutdown under all circumstances. Otherwise there is a danger that one or more processes might produce a deadlock condition, making completely automatic program termination impossible.

Figure 6.8 demonstrates the approach that the GSD debugger applies to these problems. The command to shutdown the debugger can originate from any of the user windows or from the Runtime window. Because the user windows are isolated from the debugger process,

they must be informed of the pending shutdown situation by having the windowing interface broadcast a shutdown message to all of the user windows over the IPC queues. Once this message is received by each window, it will perform the cleanup and shutdown of its respective window and then exit its program.

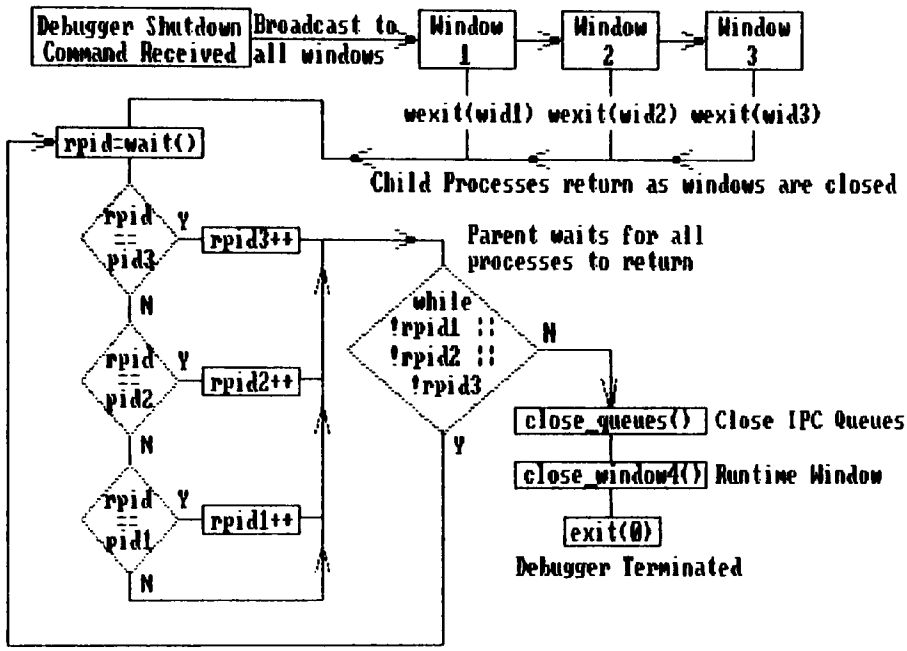

Figure 6.8 GSD Debugger and Child Process Window Shutdown Procedure

During the time that the user windows are closing down, the debugger concurrently shuts down by exiting its idle loop and then waits for the child processes to return from the window programs.

Due to the unpredictable and asynchronous timing of messages through the IPC queues, one or more of the shutdown commands could be lost if the queues are closed before all of the transmitted messages are received and processed by the user windows. This is

necessary because one or more of the returning processes may not be a window process. Instead it could be a returned debugger runtime process that had been created during controlled program execution as part of the normal operation of the debugger. This process would be killed when the shutdown command is executed by the debugger and the terminated process would then return to the parent. Therefore, it is essential that all of the window processes have returned to the parent process before closing any of the IPC queues.

To prevent this situation from occurring, a checklist is established so that each returning window process can be accounted. When a process (not necessarily a returning window process) reaches the suspended parent (blocked at the wait function), its process identifier number (pid) is assigned to the variable, rpid. This value is compared against each of the pids that were assigned at the time that the window processes were created. If there is a match, a flag is set for that respective process. When all of the window processes have returned, the checklist is completed and the parent process exits the loop. The IPC queues are then terminated and the runtime window is closed, thus completing the shutdown of the debugger.

6.3 Common Attributes of the User Windows

In the interest of not repeating the same information in the descriptions of each window, the following section is used to describe features and attributes that are common to all of the windows. Additional features of the windowing interface software will be presented in detail as it applies for a specific window or debugger feature.

The user interface of all of the windows is designed to allow input through the keyboard or function keys. Because windows 1, 2 and 3 are created using the TAM window software, they can support mouse clicks on window icons, as show in Figure 6.5. Manual command entry through the keyboard also serves to backup any mouse activated features in the event the user's mouse become defective at a future time.

Keyboard input is possible only through the window that is currently active (in foreground). It can result in either a direct debugger command, or if used in conjunction with a function key or mouse click, it can be used as supplemental data to a partially entered command. In all cases the first character entered is decoded to determine if it is a debugger command. If the command can be interpreted without additional arguments, it will be immediately passed to the debugger. For example, the character 'q' or 'Q' will be decoded as the command to quit the debugging session and will immediately initiate the shutdown sequence. If the first character decoded is not for immediate execution, the debugger will enter a 'typewriter mode' and prompt the user for more input. Upon receiving a Return or Enter character, the entered string is passed to the debugger for further interpretation.

Function keys serve to directly enter commands. Many of the hard coded function keys, such as Exit and the Up and Down arrow keys, are mapped directly to the debugger functions that would logically fit their meanings. The function keys F1 thru F8 are soft coded. The label patch above each of these keys can be redefined through software so that the same function key will have a meaning that is dependent on context of the window or menu level that the

user is currently working in. One of the design goals is to attempt to keep identical functions on the same function key to simplify operation as the user changes windows or menu levels. For example, when in the Main Menu of a window, Window Zoom is always on F8 and Window Select is always on F7.

For each window and its associated function key patches, there are two additional lines that may be written, the prompt line and the command line. The prompt line is used to advise the user of status or that additional information is needed. The command line is used to display data as it is being input 'typewriter style'. Each of the user windows has a label line in its top border that is used to identify the contents of the window.

The user windows have borders with icons that may be clicked by a mouse to issue various commands. All of the user windows have the following icons available; the Relocation Patch, Resize Patch, Up and Down Arrows and the Exit Patch. The Up and Down Arrows produce different effects depending on which of the three mouse buttons are pressed when the icon is clicked. The Left Button moves the cursor up/down one line at a time within the window. The Middle Button moves up/down one page (window size) within the window buffer with each click. The Right Button moves to the top or the bottom of the window buffer. If the mouse pointer is placed anywhere within the window and the Middle Button is pressed, the window will toggle between its current size and a zoomed size (+50% larger). The mouse may also be used to select any of the function keys, F1 through F8, by clicking within the patch regions above the function keys. In addition, if the mouse pointer is positioned on a line and the Left Button is pressed, the window will take some action that is

appropriate for that window. The specific action will be outlined for each window.

As stated earlier, each of the three bordered windows are created and controlled through function calls to the TAM windowing software. The TAM software is essentially a group of window oriented structures and specialized ioctl function calls to the kernel. Associated with each window are several structures that contain the attributes of the window and the mouse. Upon creating a window, a special status structure, WSTAT, is initialized with the x and y screen coordinates of the upper left corner of the window, the height and width of the window and the attributes of the border of the windows such as Up and Down Arrows, Border or no Border, etc. This structure can be read to determine the current values of the structure members or the application software can change the values within the structure to alter attributes of the window. An example would be changing the vertical size of the window in a zoom operation. These values of the structure can also be changed dynamically by the kernel through the use of the mouse. If the Resize patch or Relocate patch is used to change the size of the window or its location on the screen, its WSTAT structure is updated in the kernel automatically. Because each of the user windows can be set to any arbitrary size by using the Resize Patch, special software has been written to adapt the data that is sent to a window in both the number of lines vertically, and the number of characters horizontally, on a line. Each time a window is selected, resized, or moved, the UNIX kernel provides a signal (SIGWIND interrupt) to the application software to notify it that the window, and its associated status structure, has been changed in

some manner. As a result of the signal, the software evaluates the relevant window attributes and informs the underlying debugger, if necessary. It then refreshes the window to accommodate the new configuration.

Because the window can be dynamically resized along both axes, it is necessary to automatically truncate text lines in order to prevent them from wrapping around within the window. If this condition were allowed to happen, the number of lines within the window at any time would be unpredictable, making it difficult to synchronize the cursor with the debugger's text buffer. The truncation value is derived by utilizing the window size information that is contained in the WSTAT structure. Each time that the window changes size, the horizontal value is assigned to a variable that is adjusted to account for the border area. When a text message is received on the queue, its string length is compared against the truncation value. If it is smaller than the window it is passed unaltered. However, if it is larger than the available window size, it is truncated.

In order to make this feature work correctly, it is necessary to expand all tab characters to spaces in the debugger print buffer function. Also, each time that the window is changed in this way, it is refreshed to guarantee that no information is lost due to the randomness of variable re-sizing of the window.

Any of the windows can be selected into foreground in one of three ways; by clicking inside of the region of a window that is currently in background, by selecting the Window Select function key that can be found in the Main Menu of each window or by selecting the Window Manager icon that is located at the top right in the

screen of the UNIX-PC. The Default Window option of the Window Select menu is available to the user as a "cleanup" function. It automatically resets all of the user windows to their original screen position and size. Also, when a different window is selected to foreground, the command line, prompt line, and function key patches that are current for that window will be displayed.

6.4 Window 1: The Source Window

The Source Window is the first of the three user windows that is created and controlled by a child process initiated through the debugger software. Its primary purpose is to allow the user to view source documents that are associated with the program that is being debugged. The menu tree that is shown in Figure 6.9 outlines the functions of the Source Window in the following summary:

* Breakpoints: It allows the user to manipulate breakpoints at the source document level.
* Goto: It allows the user to advance to the start of a specific function, go to a specified line number within a document or perform searches on simple expressions as defined in the "ed" editor.
* Run: It allows the user to run a program in a controlled manner.
* Window Select: It allows the user to select other windows to foreground without the use of the mouse or restore the debugger windows to their default settings.
* Window Zoom: It allows the user to toggle the height of the window between 8 lines and 16 lines.
* Document Scroll: The mouse, border icons and function keys may be used to move the window up/down one line or page at a time through the text buffer or to the beginning or the end of the document.

Within the debugger, source document files may be viewed through the use of several structures. These structures are linked to the source file and buffer a portion of the text for viewing. Also associated with this is a mechanism that relates line numbers to the actual locations within the file. By issuing commands to the debugger, the user may view one or more source lines, move within a document file or change to another document file.

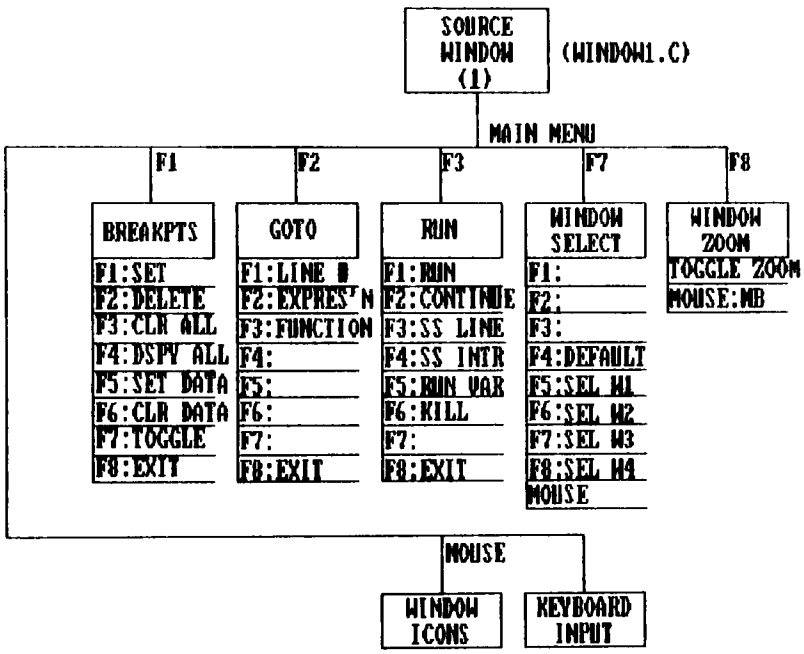

Figure 6.9 Source Window Menu Selection Tree

This process becomes considerably more complex when the windowing mode is selected. Since the Source Window is a separately running process, it cannot be tightly coupled to the text buffer as is the case when the debugger is functioning in the command line

mode. The windowing interface has been designed to maintain synchronization between the source text buffer, the contents of the window and its cursor, as shown in Figure 6.10.

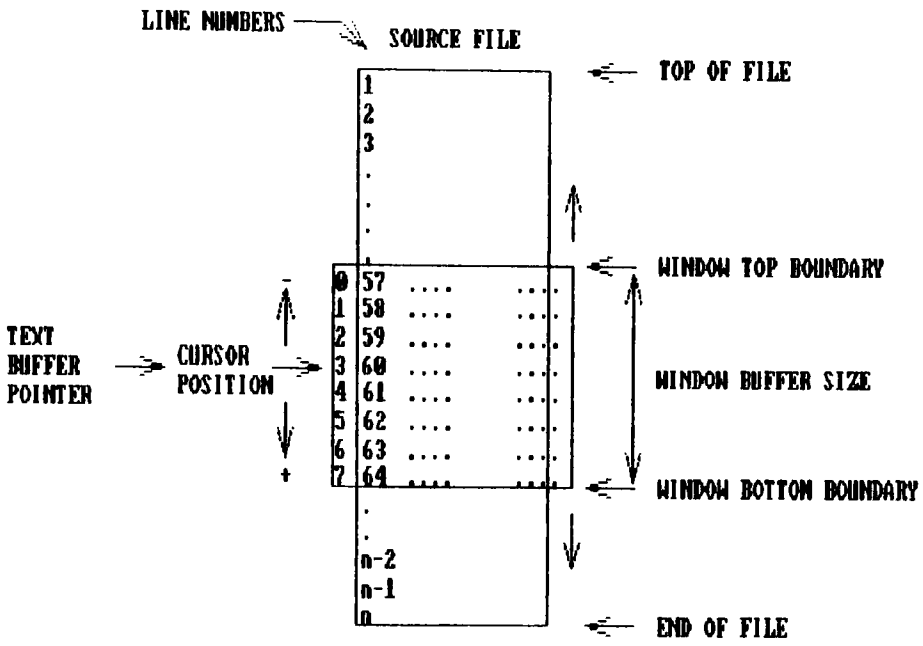

Figure 6.10 Source Window Cursor Synchronized to the Debugger Text Buffer

During the initialization of the window, its physical size is passed to the windowing interface in order to properly set the size of the text buffer. If the window changes size at a later time, due to a zoom or Resize operation, it must inform the windowing interface of its new size in order to adjust the buffer size. When the debugger has completed its initialization, it will then send one page of text (assuming a source document exists) to the window, command the cursor to the center line of the window and then update

the label line with the source file name and the number of lines in the file.

The window conceptually overlays the debugger's actual text buffer like a viewport. The cursor must stay locked together with the current statement pointer that is associated with the text buffer. Each time that the cursor is moved, the boundary conditions for the window and its current cursor position must be checked under the following rules before the cursor can be repositioned:

* If the cursor can be moved within the top and bottom border, the position of the cursor is simply adjusted within the window.
* If the cursor attempts to move down when it is at the bottom line, the new line must push the contents of the window up one line. This process may occur until the end of the buffer is reached.
* If the cursor moves up when it is at the top line in the window, the window is cleared and refreshed with the new line located at the center of the window. This process may occur until the top of the buffer is reached.

Each time that the mouse clicks an icon, or if a function key is pressed, a series of events is initiated. These events usually take the form of one or more commands or pieces of information being exchanged between a window(s) and the underlying debugger. For example, when the mouse clicks the Up Arrow icon, in order to move the cursor up one line, the following sequence of events occurs:

(Assumes that the cursor is not on a window boundary)

1. The mouse action is decoded and the debugger command "-\n" is sent over queue 0 as a message type 1 from window 1 to the debugger.

2. The windowing interface receives the message, decodes the message type and passes the type 1 command directly to the debugger.

3. The debugger adjusts the current statement pointer up one line in the buffer and then calls the function to adjust the cursor and/or window position.

4. The new position of the cursor is calculated and passed over queue 1 to the window.

5. The window process decodes the command and moves the cursor up one line position.

If the cursor had been at a window boundary, text information would be passed from the debugger to the window before the final cursor adjustment is made.

Whenever a source text line is extracted from the buffer, a breakpoint flag associated with the line structure is tested. If no breakpoint has been set at that line, the message type sent with the text will be of type 1. But, if a breakpoint has been set at that line, the message type sent will be of type 4. When the message is received at the other end of the queue and the message type is 4, that line will be output in reverse video to highlight that special situation.

In addition to positioning the cursor by means of the Up/Down Arrows or function keys, the cursor can be moved to any line within the current window by placing the mouse pointer to the desired line and clicking the Left Mouse Button on that line. The cursor will be repositioned and the text buffer will be adjusted accordingly.

As described earlier, the window may be toggled to zoom between

the vertical sizes of 8 and 16 lines. This is accomplished by writing either of the two vertical size values to the WSTAT structure. When the kernel receives the value change, it will generate a SIGWIND signal that will cause the window to inform the debugger of the size change and also to refresh the window's contents.

The GOTO menu option provides a set of functions that gives the user the ability to issue commands to go to specific line numbers or to the start of some specified function or source file or perform searches on simple expressions. All of these features are built into the debugger and are implemented in the window software by requesting the necessary additional information from the user relevant to the function selected. The window software then builds a command string in a manner similar to the document scroll operation that was described earlier.

The BREAKPTS menu option provides a set of functions that gives the user the ability to manipulate breakpoints in several different ways. Because the debugger requires that line number breakpoints be referenced to source document line numbers, it is logical to provide the breakpoint functions as one of the Source Window menu options. In Figure 6.9, function keys F1 through F4 under the BREAKPTS sub-menu control the line number breakpoint functions. To set a breakpoint, the user must position the cursor onto the desired source line and then press F1. The command is generated and sent to the debugger. When the breakpoint operation has been completed, the Source Window is refreshed to show the new breakpoint location in reverse video. The Items Window is refreshed to show the current breakpoint map. If F2 is pressed a breakpoint on the current line

is cleared, the reverse video line is cleared and the breakpoint map is updated. F3 will clear all line number breakpoints and F4 will cause the breakpoint map in the Items Window to be refreshed. Because of the manner that "-g" compiler option generates the line number section of the symbol table, not every source line may be set to provide a line number breakpoint. For more information, refer to Appendix C.

Data breakpoints are also supported by the underlying debugger. These breakpoints cause the debugger to evaluate a variable logically against another variable or constant value while the program is running. To do this, the debugger must test for the stated condition(s) on every machine instruction during a controlled program execution and halt the execution if the criteria is met. This method of operation can be **VERY SLOW** !! Pressing the breakpoint function key F5 will prompt the user to specify the conditions of the breakpoint. Once the information is entered, the Items Window will be refreshed to show the breakpoint condition and that it is currently in the "ON" (active) state. Pressing F6 will delete the breakpoint and pressing F7 will toggle the condition between the "ON" and "OFF" (inactive) state.

The RUN menu option provides a set of functions that allows the user to run the program being debugged in a controlled manner. The runtime environment is controlled by the underlying debugger and functions in the manner that is shown in Figure 6.3. The runtime functionality is very similar to the features that are available in the Runtime Window (see Window 4). Within the Source Window control a program may be run from its start, continued from a halted condition (such as a breakpoint), singled stepped either at the

source line level or at the machine language level, run until a variable changes (another very slow process) or kill (terminate) the halted runtime child process. With the exception of single stepping at the machine language level, each time that program execution is halted the contents of the window is automatically updated to indicate the position where the halt occurred. If the halt was the result of a signal, the reason for the signal will be displayed on the prompt line. The only drawback to using this window for running a program is that it cannot issue a break command to halt the program execution, either through a mouse or keyboard operation. This is because when the child process was created it was isolated from the parent to prevent its signals from affecting the parent process.

6.5 Window 2: The Items Window

The Items Window is the second of the three user windows that is created and controlled by a child process initiated through the debugger software. Its primary purpose is to allow the user to view a variety of different data displays that relate to the current status of the debugger. The specific set of displays relates to information that is usually represented in a tabular format. The menu tree that is shown in Figure 6.11 outlines the functions of the Items Window in the following summary:

* Display Functions: It allows the user to display a list of all functions used in the program and the name of its associated source file.
* Display Stack Trace: It allows the user to display the current program execution stack trace of a terminated or halted program.

* Display Machine Registers: It allows the user to display the current CPU registers.

* Display Breakpoints: It allows the user to display a list of the line number and data breakpoints.

* Display Map: It allows the user to display the address maps that can be used to relate physical memory to file offset location.

* Window Select: It allows the user to select other windows to foreground without the use of the mouse or restore the debugger windows to their default settings.

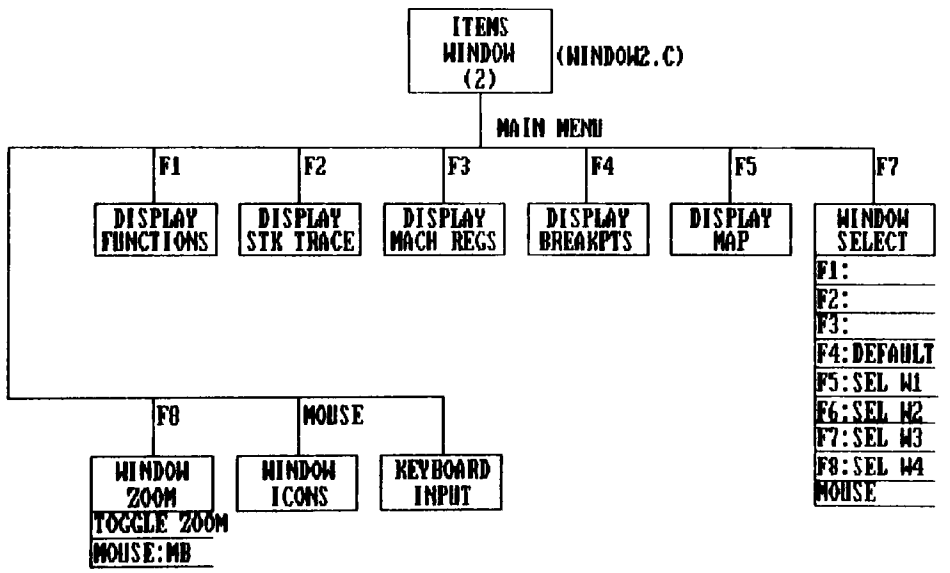

Figure 6.11 Items Window Menu Selection Tree

* Window Zoom: It allows the user to toggle the height of the window between 7 lines and 14 lines.

* Document Scroll: The mouse, border icons and function keys may be used to move the window up/down one line or page at a time through the display buffer or to the beginning or the end of the buffer.

The Items Window data buffering and display technique differs from the Source Window in several ways. It does not depend on synchronizing the cursor to a buffer in the debugger as was the case in the Source Window, but instead receives all of the information that is to be written to the window from the debugger in one block and stores it locally in a linked list. The advantage of having the data stored locally to the Items Window is that it is much easier to keep the cursor synchronized directly to the pointer of the linked list than it is to continually pass cursor offsets to the windowing interface, calculate the buffer movements, transmit the data and then send the actual cursor position information back to the window.

There is one assumption that the buffering technique of this window falls under. It is that there will probably not be large quantities (a hundred lines at most) of data passed to the window. The register dump and memory map are very small tables of fixed size and the breakpoint table will probably never be over 10-20 entries at any time. The stack trace may get large if the program being debugged has a lot of recursion but the largest entry will probably be the list of program functions. Even if all of the source documents have, for example, 200 functions, it should not seriously tax the resources of the machine. Given the above assumption, the Items Window process will only create linked list nodes on an as needed basis. When the window is refreshed, the

existing linked list nodes are not destroyed but are "erased" by clearing a data valid bit before the new data is sent. Upon receiving new data, if there are an insufficient number of nodes available, additional ones will be created. This process saves time as memory allocate/de-allocate functions are rather slow.

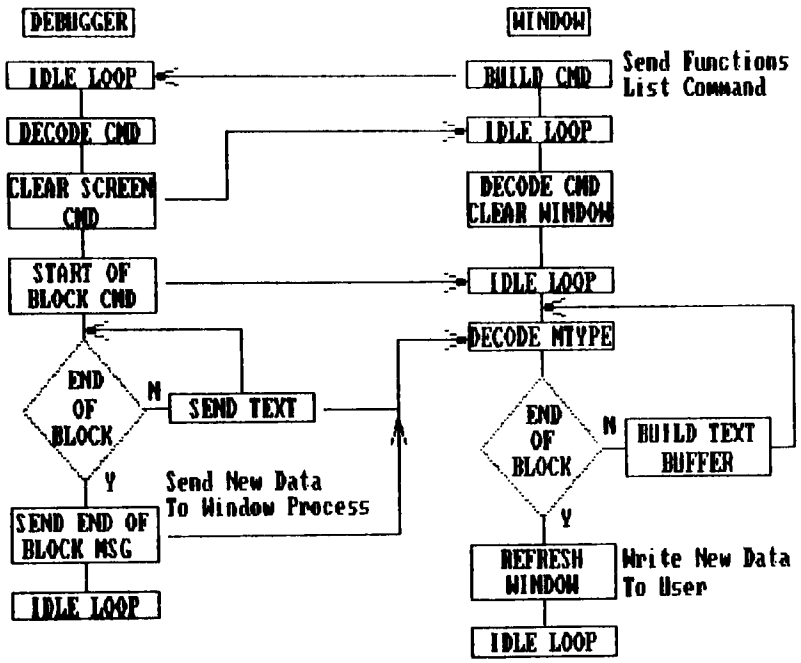

Figure 6.12 Debugger to Window Data Block Transfer Scheme

As described earlier, all data is transferred from the debugger to the window in block transfer fashion. The example shown in Figure 6.12 depicts the following series of events that are set into motion if the Send Function List command is selected:

* The window issues a command to "Send Function List".
* The debugger decodes the command and issues a "Clear Window" command (See Appendix B) to the window.

* The debugger then sends a Start of Block Transmission message to the window process over the IPC queue.

* The window receives and decodes the message and prepares to receive data by first flushing the existing linked list of data by clearing the "free" flag in each link node. Then it enters a loop to receive data until an End of Transmission message is sent. As each data line is received by the window process, it is entered into a linked list node.

* When the debugger has finished its transfer, it sends an End of Transmission message to the window over the IPC queue.

* Upon receiving the End of Transmission message, the window process exits its loop and writes the first window full of data for the user.

Once the data has been transferred to the text buffer of the window, the synchronization of the cursor to the buffer pointer is managed in much the same way as it is in the description of the Source Window (see Figure 6.10). Now instead of the source file and its associated text buffer the data is stored in a linked list that is local to the window. Also, the same set of mouse and function key commands as well as the window boundary conditions apply.

Normally, the Items Window has icons on its border to allow the user to vertically scroll data within the window and resize the window. The Display Machine Registers and Display Map selections are always a fixed size and therefore do not have the Vertical Scroll and Resize patches. Instead, when selected, they will automatically be set to the correct size for their respective contents. Also, these windows will not have the ability to zoom either through the function keys or the mouse.

The Display Functions menu option allows the user to obtain a listing of all functions that are contained in the source document(s) along with the name of the file that the with which the function is associated. This feature derives its information through a scan of the procedure table by the debugger. One of the members of the procedure table structure is a flag that is cleared if a function name is found in the symbol table of a compiled source program. It is this flag that distinguishes a function in a source program from a built in function (e.g. one that is part of a library). By scanning the symbol table and noting the state of the flag, all functions that are included in the source files may be collected for the display buffer of the window. In addition, the procedure table also contains a pointer into the file table to identify the source file that the function is associated with. The formatted data is then passed in block form through the queue to the window.

The Display Stack Trace menu option allows the user to display the current program execution stack trace of a terminated or halted program. The trace is created by evaluating the user stack area from the current stack pointer working backwards to the end of the stack area. The display includes function calls and its optional argument list, as well as all variables that are pushed onto the stack along with its current value for that position in the stack. In addition, the name of the source file and the location line number is given. The stack search can become lengthy if a function is called recursively. For more information on stack organization, refer to Appendix D. If the windowing mode is enabled, the formatted data is then passed in block form through the queue to the window.

The Display Machine Registers menu option allows the user to display the current contents of the CPU registers of the executing program process. This information is obtained from the user area of memory through ptrace function calls. If the windowing mode is enabled, the formatted data is then passed in block form through the queue to the window.

The Display Breakpoints menu option allows the user to display a list of line number and data (conditional) breakpoints. Internally the debugger stores the breakpoints as a linked list of structures. Each of these structures contain the following information about the specific breakpoint: the memory address of the breakpoint, the machine instruction that was stored at the breakpoint location (prior to the insertion of the trap instruction), source line number of the breakpoint, a flag to denote whether the breakpoint is a line number or conditional breakpoint, a flag to denote if the breakpoint node is still valid (for the linked list), text string of a conditional breakpoint, procedure name of the breakpoint and the pointer to the next link node. When the command to display breakpoints is issued, the debugger scans its internal list of breakpoints, first for the line number breakpoints and a second time for the data breakpoints. If the windowing mode is enabled the formatted data is passed in block form over the IPC queue to the window.

The Display Map menu option allows the user to display the address maps that can be used to relate physical memory to the core and object file offset locations (see Appendix A). This map is created by the debugger during its initialization phase and is stored in a map structure. When the display command is received and

if the windowing mode is enabled the formatted data is passed in block form over the IPC queue to the window .

6.6 Window 3: The Variables Window

The Variables Window is the third of three user windows that is created and controlled by a child process initiated through the debugger software. Its primary purpose is to allow the user to display values of local and global variables or assign values to them. The menu tree that is shown in Figure 6.13 outlines the functions of the Variables Window in the following summary:

* Display Locals: It allows the user to display a list of values of all of the variables and their addresses that are local to the current function.

* Display Globals: It allows the user to display a list of values of all of the variables and their addresses that are either external or static types not declared inside of a function.

* Display All: It allows the user to display a list of both the Local and Global Variables, as described above.

* Assign Variable: It allows the user to write a new value to a specified variable. This can only be done if the W-flag is toggled to the set state (see the Runtime Window for more information).

* Window Select: It allows the user to select other windows to foreground without the use of the mouse or restore the debugger windows to their default settings.

* Window Zoom: It allows the user to toggle the height of the window between 7 lines and 14 lines.

* Document Scroll: The mouse, border icons and function keys may be used to move the window up/down one line or page at a time through the display buffer or move to the beginning or the end of the

buffer.

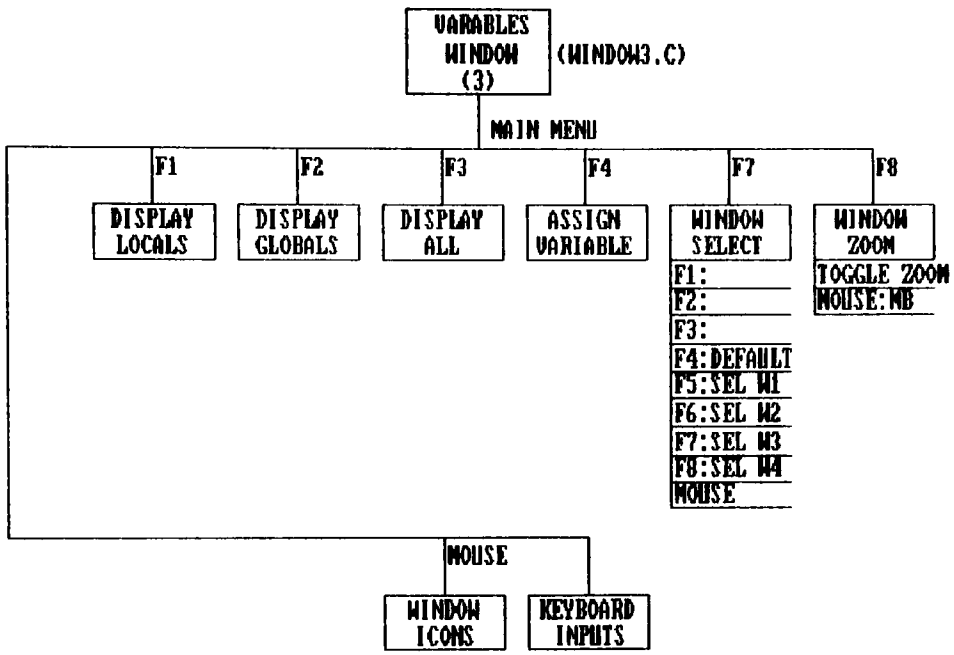

Figure 6.13 Variables Window Menu Selection Tree

The Variables Window data buffering and display technique is the same block transfer method as was previously described in Section 6.5, the Items Window. However, the process of deriving and displaying data values is a very different and complex process. The exact procedure for recovering the value for the variable will not be discussed in detail as it is outside the scope of this paper. The discussions will examine how the variable types are discriminated and returned to the window, how the information is formatted and presented to the user and how variable values may be updated.

The issues of relating a value to a given variable is the most complex process in the underlying debugger. A few of the factors that must be considered in the resolution process include:

* Is the variable global or is it local to a specific procedure?
* Is the variable a simple type (e.g. integer, char, etc.) or is it a derived type (e.g. pointer, array or function.) ?
* If the variable is of a derived type, how should it be represented to the user ?

When the Variables Window is initialized during the start-up process, it is set to receive the local variables of the procedure that it is currently in. This initial mode was chosen over the global listing on the assumption that the total number of local variables would most likely be small in relation to a presumably larger global variable list. This would lessen the time delay impact during the debugger start-up process. If no core file is present or if a runtime process has been terminated, a message will be sent to the window to advise the user that no information is currently available.

The commands used to obtain the lists of global and local variables set in motion a series of filtering algorithms that are applied to the symbol table. Because each of these lists depend on different information in the symbol table, it is instructive to examine these evaluation techniques separately.

Local variables can only be evaluated for procedures that are on the user stack (active procedures). The local variable list is compiled and presented to the window through a series of information

interchanges between the Variables Window and the underlying debugger. When the Display Locals (F1) function key is selected, the window manager issues a command to the windowing interface for decoding. The interface returns a start of block marker to the window to prepare it to receive the pending data transfer. If there is no core file or active process, a warning message is returned to the window and the data transfer is concluded, otherwise, the **find_locals** function, shown in Figure 6.14, is called.

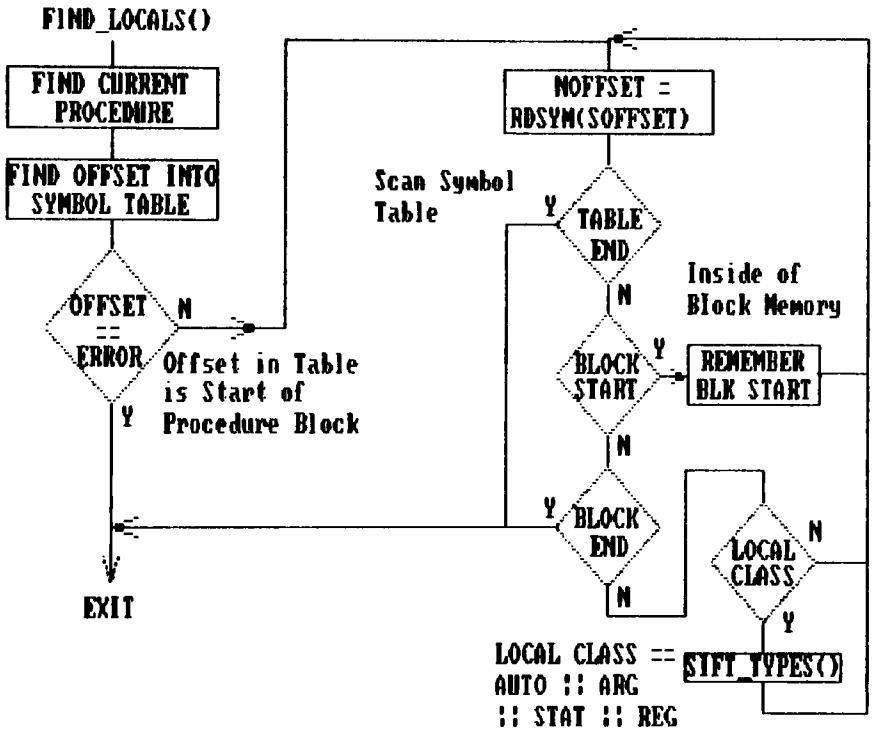

Figure 6.14 Symbol Table Filter for Local Variables

To obtain a list of variables for the currently active procedure, it is first necessary to determine which procedure the debugger "program counter" is currently in and assign a procedure pointer to it. Within the symbol table, the procedure is organized

as a block with a recognizable beginning and end. All symbols that belong within the scope of the procedure are contained within the block in the same order as they appear in the function argument list or the declaration list. The procedure pointer can then be used to point to the start of the specific procedure block within the symbol table. The table is scanned for symbols with the proper storage classes (e.g. auto, argument, static and register) until the matching end of the block marker is found. As each symbol (one that is a local variable) is found, it is passed to another function, sift_types (see below), for further processing and formatting. The end of block message is passed to the window to be refreshed with the new data.

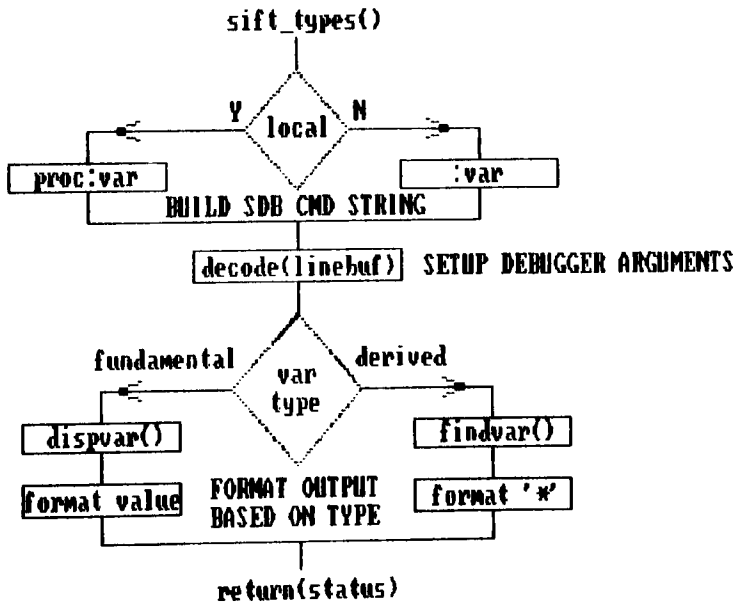

Figure 6.15 Symbol Table Filter for Types

The global variable list is compiled and presented to the window through a series of information interchanges between the Variables Window and the underlying debugger. When the Display Globals (F2) function key is selected, the window manager issues a command to the windowing interface for decoding. The interface returns a start of block marker to the window to prepare it to receive the pending data transfer.

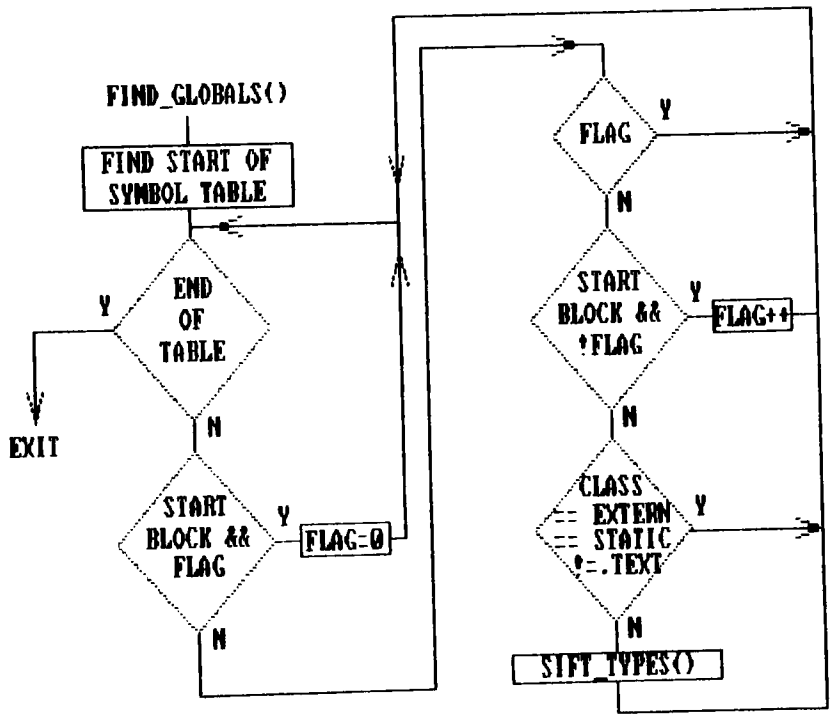

Figure 6.16 Symbol Table Filter for Global Variables

If there is no core file or active process, a warning message

is returned to the window and the data transfer is concluded, otherwise, the find_globals function, shown in Figure 6.16, is called. Unlike the local variables, global variable declarations are not contained within the scope of a block such as a function. Instead, they may be declared in multiple header and source files that are not resolved until link time. Because of this "scattered" nature, the entire symbol table must be scanned when searching for global variables. The process begins by assigning a pointer to the start of the symbol table. If a start of block marker is encountered, the entire block is skipped over, because these symbols are hidden as per the scoping rules. If a symbol has a storage class of external or static, and it is not of the .text section (e.g. a function name), it is passed to the sift_types function (see below) for further processing. The pointer moves through the symbol table until the end of the table is reached. The end of block message is passed to the window to be refreshed with the new data.

Sift_types builds a sdb command to evaluate a variable from the symbol name that is passed from either the local or global variable filter functions that were described earlier. If the symbol is a global variable, the sdb command will have the format;

:symbol name/\n

If the symbol is a local variable, the sdb command will have the format;

procedure name:symbol name/\n

Once the variable is evaluated, the next step is to determine how its data is to be represented to the user. This decision is made based on the storage type of the symbol.

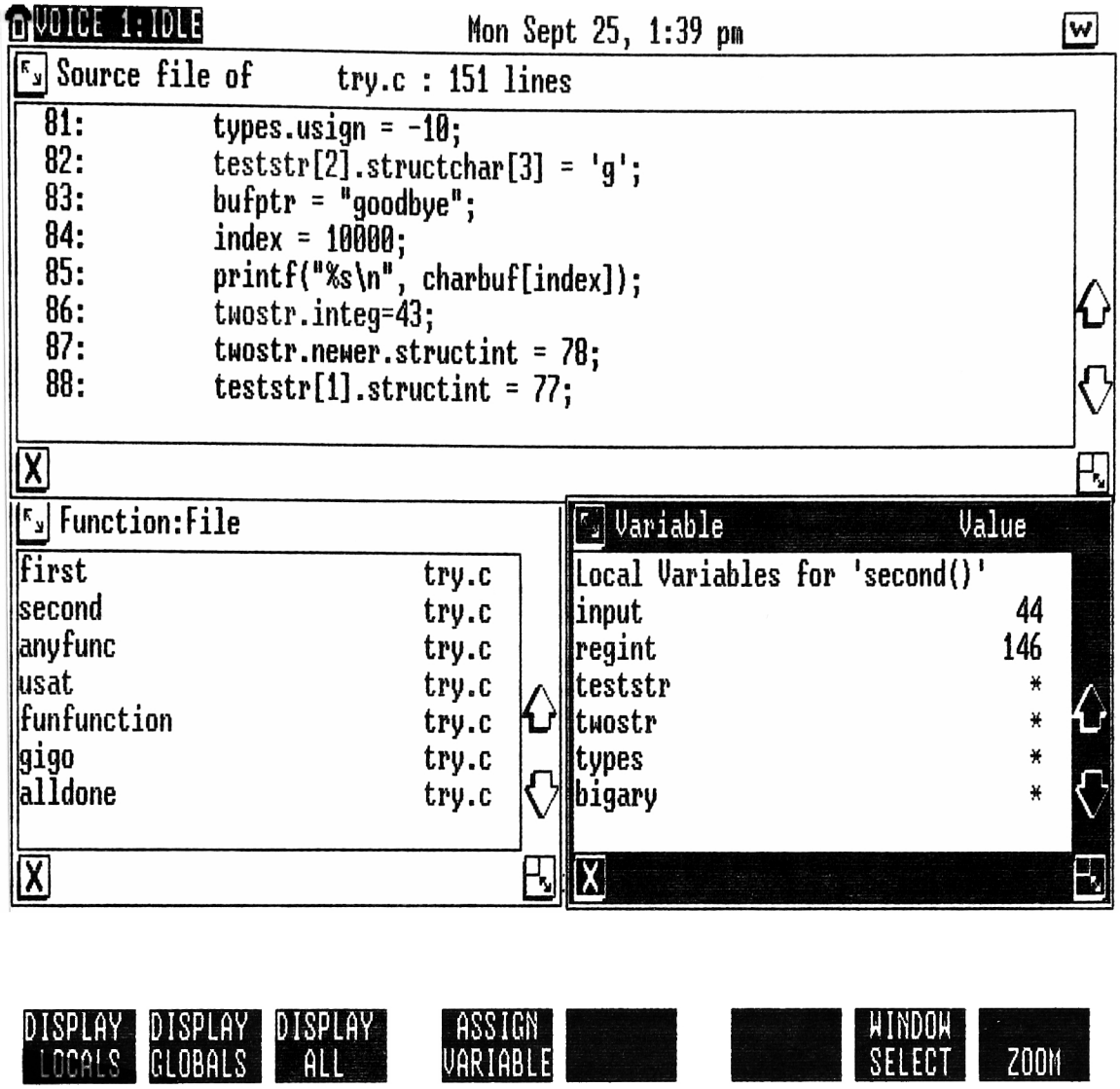

Figure 6.17 Variables Window Display

If the symbol type is fundamental (e.g. char, int, long, etc.), the value is formatted into a buffer with the variable name and its associated address before it is passed to the window. If the symbol type is derived (e.g. array, function or pointer) an asterisk '*' is substituted in place of the value to denote that it can be expanded at a later time.

Once the end of block marker is received by the window manager, the display is refreshed with the new data. Figure 6.17 shows the windows in a typical display situation.

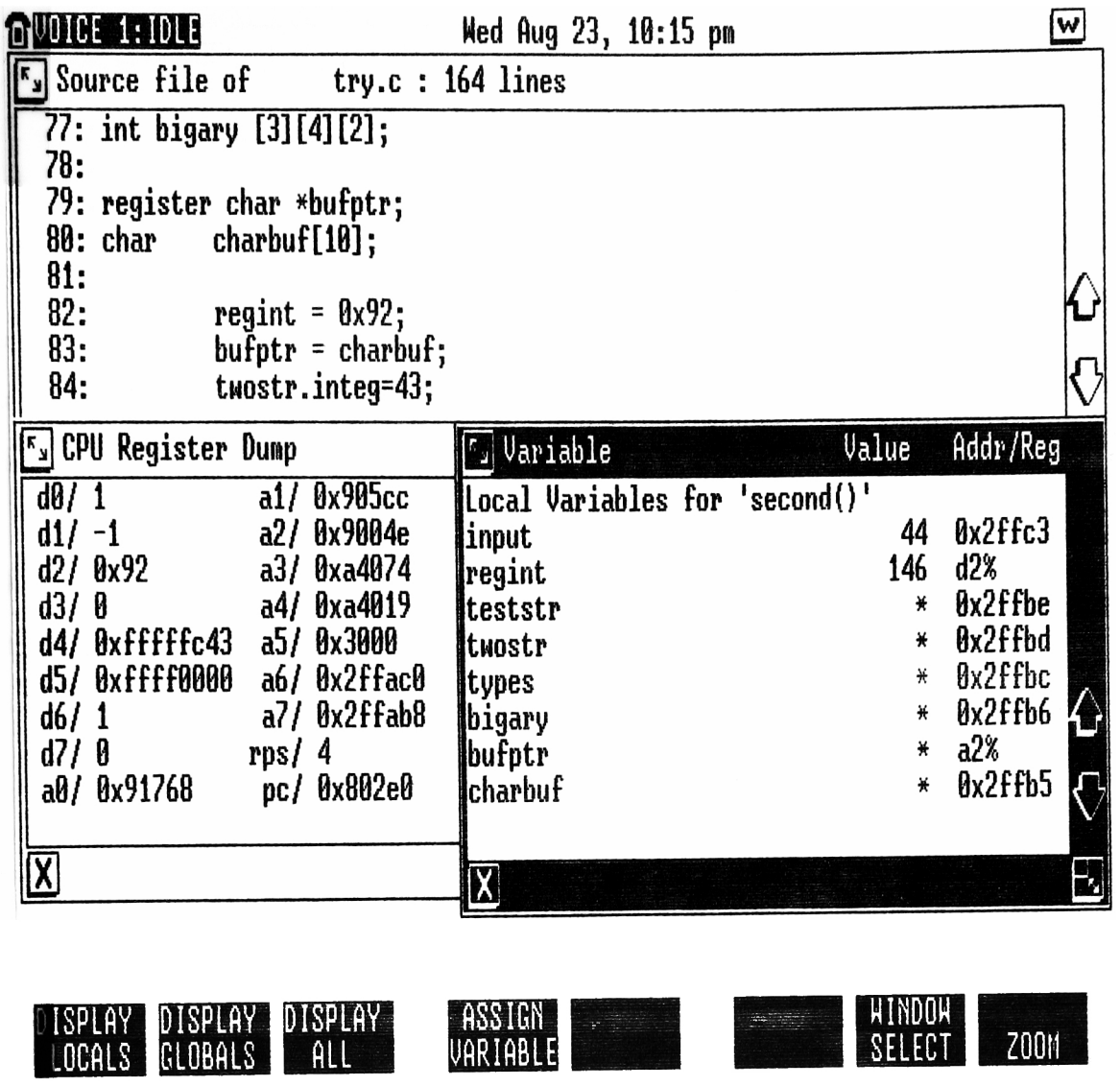

Figure 6.18 Window Expanded to Show Hidden Address/Register Column

The Variables Window is showing the list of local variables for the function, **second**, that appears in the Source Window. Two of the variables, **input** and **regint**, are of fundamental type and have their current values displayed. The other variables are derived with their respective values represented by '*' to denote that distinction. Each line in the window also contains the address or register number that is associated with the variable. This information is in a column that is hidden when the window is in its default size but it may be viewed by expanding the right border of the window as shown below.

Figure 6.18 depicts the Variables Window expanded to show the hidden address/register information column. This can represent three different kinds of information; as an absolute address in the .data/.bss section (0x9XXX), as an address location on the stack (0x2XXXX) or as an address or data register of the CPU (e.g. a2%, d7%). The Items Window has been set to show the CPU Registers. Some of these are register variables in the Variables Window. The following are some examples of how this information can be interpreted:

VARIABLE	VALUE	LOCATION
input	44	value at 0x2ffc3 on stack
regint	146	register d2 contains 0x92 = 146
bigary	array	start of array at 0x2ffb6 on stack
bufptr	pointer	register a2 has address 0x9004e, start of buffer in .data section

Table 6.2 Interpretation of Variable Value Based on Storage Type

As stated earlier, variables that are not fundamental types will not have their values directly displayed. This is done for the sake of simplicity in the window and to save time when the window buffer is being created. One of the features of the MacMETH debugger, as described in Chapter 2, is the ability to expand derived variables in order to view each of its individual members. If one of the expanded members is itself derived, it would also be represented by a '*', as was its parent structure. It could then in turn be expanded. This process could be continued almost indefinitely so that structures, such as a linked list, could be traversed by successively expanding the pointer to the next structure.

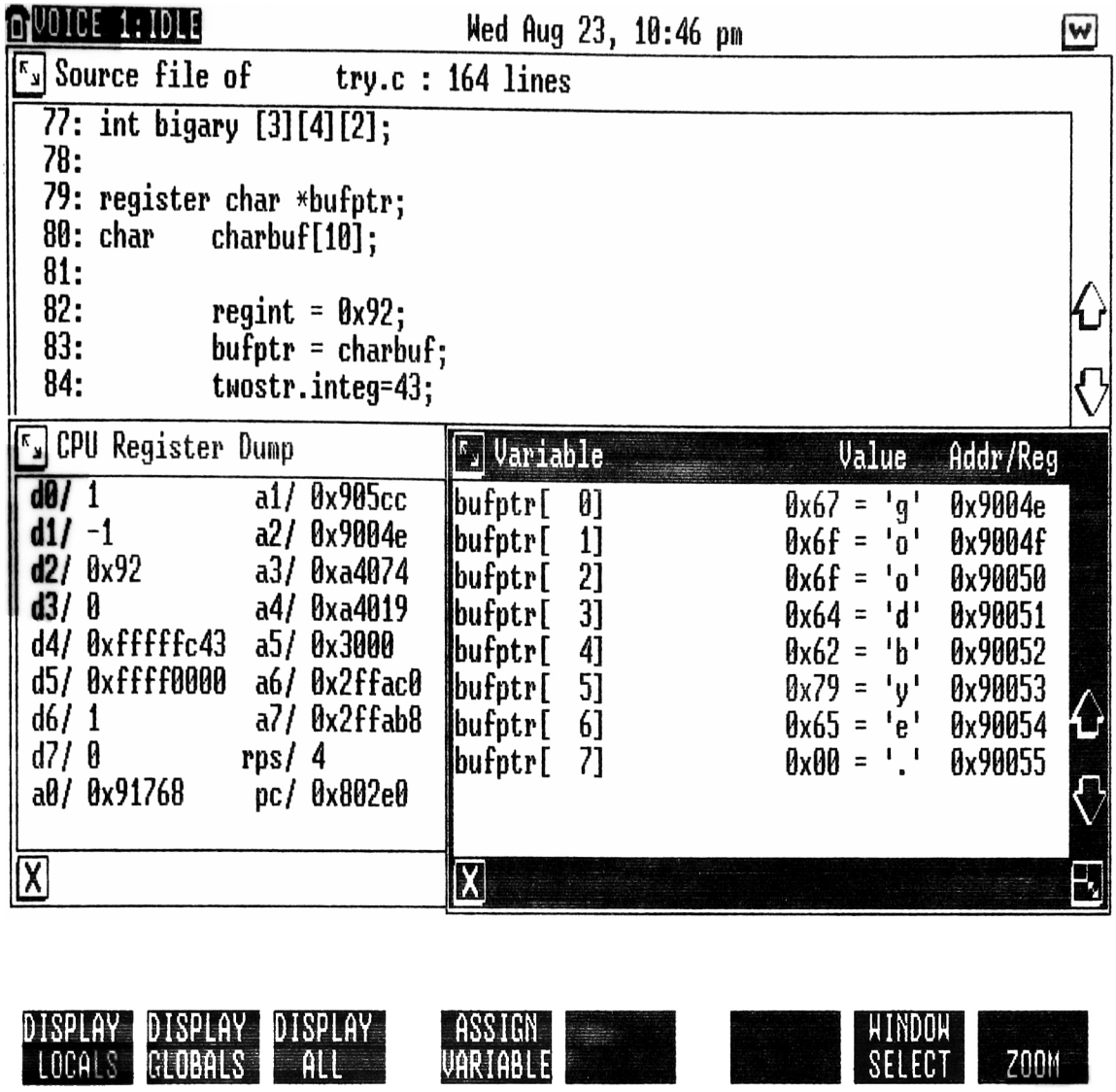

Figure 6.19 Array Expanded to Show Individual Members

The underlying SDB debugger has a limited capability to expand derived variables manually. However, it was not originally designed to perform this task on multiple levels, remotely, as was the MacMETH debugger.

The question then becomes how to exploit the existing variable expansion capabilities of SDB from a remote process window. A method of expansion was designed to work on a derived variable to allow the user to "look" down one level. A variable that is a single derivative (e.g. a pointer, function or array) will completely expand. However, the COFF format allows variables to have up to six

levels of derivation. An example of a variable with two derivatives would be an array of pointers. This expansion technique will also work on successive expansions if the variable name remains the same. Figure 6.19 shows the expanded character array that is pointed to by **bufptr**.

In a manner consistent with other features of this window, the expansion process is a series of data exchanges between the window manager and the underlying windowing interface and debugger. The following is the sequence used in the expansion process:

* When a line is clicked for expansion, the window manager software packs the line into a buffer by removing all white spaces. This buffer is then sent to the windowing interface tagged with the command to expand the variable (that is, if it is a variable that can be expanded).
* The interface software parses the packed line to extract the variable name and to determine if it is a local or global variable. If it is local, the symbol table is scanned within the specific function block to set the pointer. If not, the procedure is altered to search for the variable as a global.
* The symbol type is evaluated to determine if it is fundamental or derived. If it is derived, the fundamental type and first and second derivative values are extracted for the expansion decoder.
* The derivative decoder builds an SDB command based on the fundamental and the derivative types and appends it to the variable name.

* The command is passed to the debugger. The resulting data is passed back to the window in block form.

The exact method of how variables and their respective data are represented are under the control of the debugger mechanism. Note in Figure 6.14, even though the declaration of the array of characters, **charbuf** is of size 10, in the Source Window, only the first 8 elements are displayed. This is because the debugger will only display the elements until a NULL (0) character is reached. This could be a limitation if the data in the array was of binary data, not ASCII characters. The occurrence of a zero as data will prematurely terminate the display of the entire array. If the size of the array dump is explicitly given, the array would be displayed regardless of the data content. However, if no limit size is explicitly given and no NULL character is detected in the data, the debugger will automatically stop at 100 elements.

This method of variable evaluation and expansion could be improved but it would involve a major redesign of the existing debugger. This is beyond the scope of this thesis project.

The last feature of the Variables Window is the Assign Variable function. It allows the user to assign a new value to a variable. When the user selects the F4 key, a prompt will appear requesting the variable name (or member description if it is a derived variable type) and then the new value. The window manager will assemble this information into an SDB command and pass it to the debugger where it will be processed. One requirement of this feature is that the W-flag be set to enable writing to the core file. If the flag is not set, an error message will be displayed. This flag toggle can be controlled through the Runtime Window.

6.7 Window 4: The Runtime Window

The final major aspect of the underlying debugger is the mechanism through which the program being debugged can be executed in a controlled manner. The menu tree (Figure 6.20) outlines the functions of the Runtime Window in the following summary:

* Run: It initiates a controlled program execution from the start of the program. The program will execute until it terminates normally or is halted by a breakpoint or interrupt.
* Continue: It continues a program execution after a breakpoint or an interrupt.
* Single Step One Source Line: It causes a program to execute the statement in the current source line.
* Single Step One Machine Instruction: It causes a program to execute the machine instruction at the current program counter location.
* Step Until a Variable Changes: It causes a program to single step until a specified location is modified.
* Kill Process: It terminates the current program's executing process.
* Miscellaneous: It allows the user to perform housekeeping functions. At present only the W-flag may be toggled to control writing to the core file.
* Window Select: It allows the user to select any or all of the three user windows back to foreground.

Unlike the prior three user windows, this window is not a separately running process. Instead, it is created as a separate

I/O device that is associated with the underlying debugger process. Now the debugger process can directly accept user keyboard input.

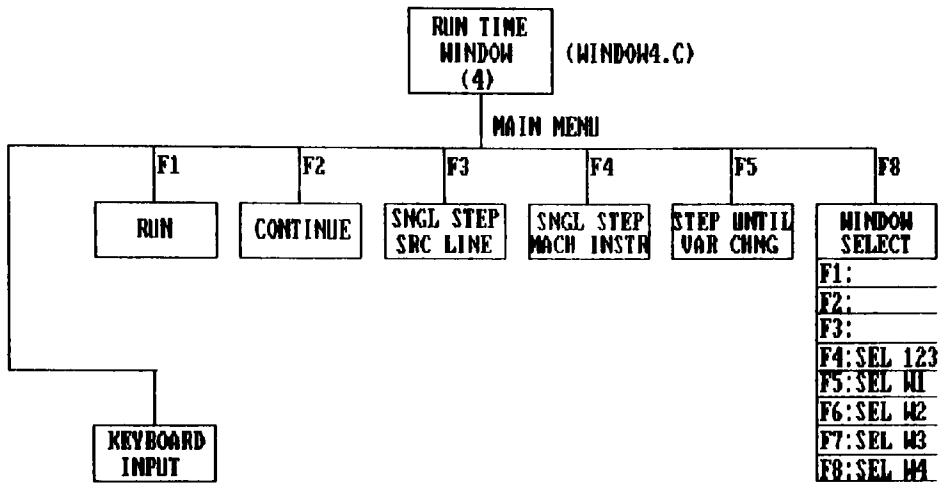

Figure 6.20 Runtime Window Menu Selection Tree

This closely coupled relationship is necessary for one reason, so that keyboard commands pass directly to the debugger and not through an IPC queue. Otherwise a keyboard interrupt (^C) would not be detected by the debugger as a signal to suspend the execution of a process. A keyboard interrupt that is generated by one of the user windows will not halt an executing debugging process. This is a shortcoming of the RUN function that is found in the Source Window command structure. The actual runtime mechanism is outlined in detail earlier in Section 6.1, SDB: The Underlying Debugger.

The attributes of this window also differ from the user windows

in several ways. As seen in Figure 6.21, it is created as a full screen, borderless window so that during the execution of a program being debugged no information would be lost or distorted due to wraparound that might occur in a smaller window environment. The keyboard and queue polling is similar to the method in the user windows, shown in Figure 6.4, except that when the Runtime Window is selected the polling of the IPC queues is skipped with the process always blocking on keyboard input.

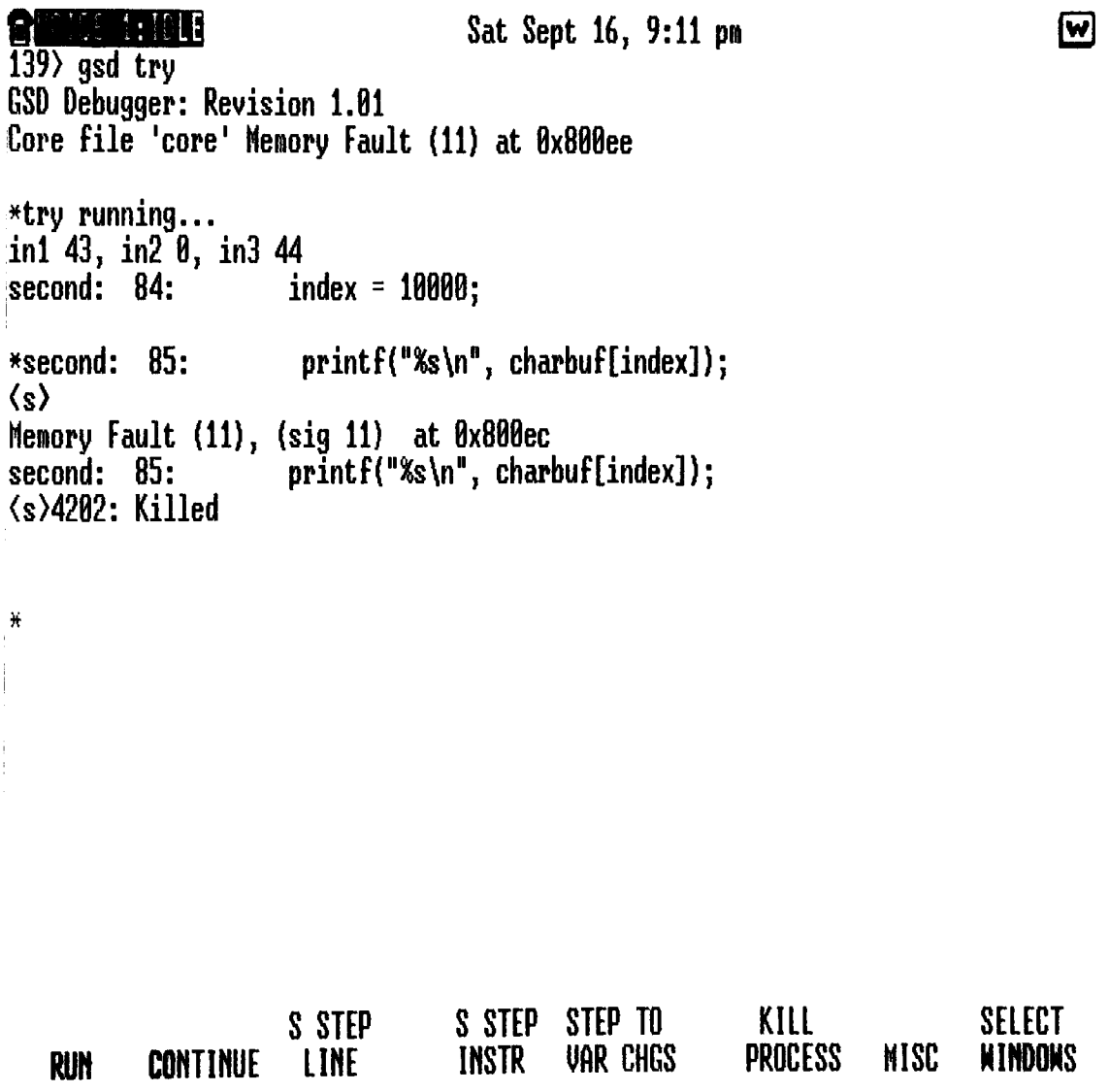

Figure 6.21 A Typical Runtime Window

There is no need to poll for queue input as there never can be any from the user windows when the Runtime Window is selected.

There is no need to poll for queue input as there never can be any from the user windows when the Runtime Window is selected. The Runtime Window menu provides a set of functions that allows the user to run the program being debugged in a controlled manner. The runtime environment is controlled by the underlying debugger and functions in the manner that is shown in Figure 6.3. Within the Runtime Window menu there are a number of control options available.

The RUN command initiates a program process from its start. All control pointers are initialized and a child process is forked and executed under the control of ptrace until such time that a breakpoint or signal halts the process, returning control to the parent debugger.

The CONTINUE command, restarts a process from a halted condition such as a breakpoint or an interrupt. The process once again runs under the control of ptrace until halted.

A halted process also may be SINGLE STEPPED at either the SOURCE LINE level or at the MACHINE LANGUAGE level. This execution mode is under the control of ptrace, but it does not depend on a breakpoint or interrupt to halt the program execution. If the stepping is performed at the machine code level, the machine instruction will be output in the Runtime Window.

RUN until a VARIABLE CHANGES (another very slow process) executes single machine instructions, and then checks to determine if a specified variable has changed. This process is repeated until the specified variable changes value. At that time the program is

halted.

The KILL command terminates the halted runtime child process and all other support components of the debugger that are associated with its controlled execution such as breakpoints, stack pointers, etc.

With the exception of single stepping at the machine language level, each time that program execution is halted, the contents of the Source Window are automatically updated to indicate the position where the halt occurred. If the halt was the result of a signal, the reason for the signal will be displayed on the prompt line.

The Window Select function can select any one of the three user windows back to foreground or request that all three return to foreground together.

7. Conclusion

Writing a practical debugger is as much of a study in user esthetics as it is in how to accomplish the task of writing the code to access the program data. The problems that are encountered when converting a traditional command line oriented debugger into a windowing environment provide stimulating challenges over a wide range of disciplines. This chapter will explore the outcome of the project from several different perspectives. The end result will be to qualify the final product against the original conceptual design.

As discussed in Chapter 2, Comparison of Available Debuggers, SDB represents one of several debuggers that are commercially available. Its choice as the underlying debugger in GSD was made because it was the symbolic debugger on the 3B1 computer and because its source code could be obtained in order to make modifications as needed to develop the new debugger. In addition, the 3B1 supports the TAM window package that would be the basis of the user interface. Using SDB as the underlying debugger and building the user windows and command interface on top with TAM seemed to be a reasonable solution to the required task.

7.1 Advantages of the GSD Debugger

The primary design goal of GSD is to provide an easy to use debugging environment through interactive multiple windows. These windows and their associated functions can be selected through mouse or keyboard function keys. This represents a significant improvement over the original SDB debugger that required the knowledge of a rather cryptic command set. The user was further hindered by the absence of good quality documentation. With GSD,

the user is provided with continuous access to multiple sets of information to aid in the debugging process. Debugging a program is difficult enough without struggling with a software tool that is supposed to help, not hinder the user.

The debugger interface is designed so that related functions are grouped together with an appropriate window. For example, the Source Window has several functions that require referencing the source document in its command set; such as String, Line and Function Searches and Breakpoints manipulation. The overall menu architecture is relatively flat so that the user is not required to remember long sequences in order to build a command. All commands that can be selected through the mouse are backed up with redundant keyboard function keys or command sequences.

In addition to the window oriented mode, GSD can also be invoked in the traditional command line oriented mode by setting a command line flag.

7.2 Problems Encountered, Known Limitations and Known Bugs

Overall functionality of the project design was limited by shortcoming in the underlying SDB debugger, the TAM window package and the UNIX operating system of the 3B1.

SDB was written to debug single process programs and in itself cannot be easily modified to debug multiple process programs. Also, it cannot process signals while debugging in controlled execution mode. The 3B1 implementation of SDB can set conditional breakpoints in the source code, However, there are bugs in the code that prevent it from functioning properly. Some corrective measures have been taken, but there is still work to be done before the command can be fully functional.

The TAM window package is not well documented. The documentation exists contains a number of errors (including documentation for TAM related functions, such as ioctl functions in window(7)). One of the greatest limitations in TAM, on the 3B1, is when multiple windows are created from a single process only the window that is currently in foreground can be updated. If a window to be updated is in background, it must first be selected to foreground. This window selection process has the undesirable effect of creating a lot of annoying "flashing" on the screen. To eliminate this problem necessitated implementing the user windows as multiple processes. In this manner, a window can be updated whether it is in background or foreground. Even with this technique TAM does not allow the label line at the top of the window to be updated if the window is currently in background.

Another TAM related problem is periodically encountered when a bordered window is moved or resized. Any time that the window is modified, a window signal (SIGWIND) is generated by the kernel to inform the application software that the window has changed in some manner. During that time the data structure containing the physical attributes of the window in the kernel periodically becomes altered. When this happens, the current window changes into a full screen borderless window. This bug cannot be directly fixed. Instead, each time that a window signal is received the underlying structure must tested against an image template and reset to the correct values if it has been corrupted. This phenomenon has also been observed in the user agent office environment several times.

TAM does not easily support window functions for remote terminals (even between 3B1 systems). If the remote device is not

TAM compatible, the graphics are "emulated" through curses. In addition, remote mouse controlled input is never supported by TAM. Because of these limitations the design of GSD is limited to debugging locally at the console.

The operating system of the 3B1 allows a maximum of ten windows devices to be open at any time. Since GSD uses four windows (the debugger window and three user windows), care must be taken no to have more than a total of six windows open at the time that the debugger is invoked. Otherwise it will not be able to open all of its own windows and will shut down. This limitation also prevents more than one instance of the debugger from being run at any time.

The debugger communicates to the user windows through IPC queues. The windows and queues are created when GSD is invoked and deleted when GSD is exited normally. If the debugger crashes without exiting normally, the windows and queues may not be closed and must be manually terminated. This situation can be corrected by initiating GSD through a separate process. In this way, when the debugger process returned, its status would be checked. If there was an error, the initiating process could perform a cleanup operation.

Portability of source code between different computers is desirable whenever possible. The SDB debugger is very machine dependent and has to be tailored to the specific system architecture in order to compile and function properly. The TAM window interface might be portable between different computers if the header files and libraries are kept compatible. However, the only way to test portability is to compile the code and resolve any problems that occur on a case by case basis.

7.3 What Was Learned

During the initial phase of the project the source code to the SDB was not available. It was therefore necessary to do the initial work by porting the source code of SDB that was available for the 3B2 computer. Resolving machine dependencies between the 3B1 and 3B2 versions of SDB proved to be a great challenge as well as a good learning tool. In order to complete the code porting, it was necessary to gain intimate knowledge of nearly all aspects of the construction of SDB. This became especially useful when the design of the windows and the debugger interface was begun. In addition to the design details of SDB, other areas related to the UNIX environment, such as the makeup of the core file, had to be investigated. This required the writing of special software to probe both core and executable files in order to dissect and document them.

The implementation of the TAM window package proved to be equally challenging. This was in part due to poor documentation, several performance limitations and an insidious bug in the kernel that eluded detection for several weeks. The initial attempt to implement the three user windows was done as a single process in the debugger. This worked fairly well except that a window had to be brought to foreground before it could be updated. The effect was unpleasant flashing on the screen each time a new window was selected for updating. The solution was to implement the user windows as separate processes apart from the debugger.

Writing the debugger and windows as separate processes open many avenues for practical display updating and access coordinating. However, because the user windows are no longer tightly coupled to

the debugger it becomes necessary to pass information over interprocess queues. To accomplish this, a simple protocol and source/destination scheme was devised. This technique also provided some interesting methods through which to synchronize operation between the windows and the debugger. The solution of using multiple processes for the debugger and its window also brings with it the matching implementation challenges. Several design cycles were incurred before the start-up, run and shutdown mechanisms were functioning properly.

Besides all of the technical implementation issues that were involved, designing the user interface to be functional and yet easy to use was no trivial matter. Several trade-offs had to be made with respect to overall functionality because of design limitations imposed by SDB, TAM or the user facilities of the 3B1 (e.g. number of window devices that could be open at any one time). Debugger functions were logically grouped together within a particular window. One major objection to the original SDB was that it had a cryptic and difficult to use command set. In GSD, many of the SDB functions are initiated through a single function key or the mouse. When necessary the user is prompted to enter additional input thought the keyboard.

7.4 GSD: The Outcome vs the Initial Expectations

In general the end product met or exceeded the initial conceptual design goals:

* Design an interactive, multiple window interface onto an existing debugger.
* Provide the user with constant access to

multiple pieces of information about the program being debugged.

* Free the user from a cryptic command set and replace it with a mouse driven set of icons and function pads. All debugger commands are backed up through keyboard function keys or manually entered debugger commands.

Forward progress was often hindered by problems (discussed earlier) to which there were not readily available answers or erroneous information. However, when one considers that areas of UNIX were explored in which there is no documentation or persons to turn to for help, it is a minor miracle that the end result was accomplished. Finding solutions to problems not only resulted in the accomplishment of the final product but also taught the designer many things that would not have otherwise been learned (that's what it is all about, isn't it).

7.5 The Future of GSD

Part of the impetuous for creating GSD was to provide an alternative for the rest of the world (at least with respect to the 3B1) to the line oriented SDB debugger. This has been accomplished. The initial proving ground for this software will be the Computer Science Department of Rochester Institute of Technology, for whom this thesis and associated software has been written.

Outside interest has already been expressed for a similar debugger to run on a 3B2 computer using X-Windows instead of TAM. This project may someday undertaken.

There is a need for window oriented debuggers such as GSD and those that are capable or supporting work on multiple process to be

made available in new releases of UNIX. However, until the industry is willing to invest in the effort, it will not easily happen. To put it in the words of Thomas Cargill of AT&T Labs, creator of the Blit and Pi debuggers, "Writing debuggers is considered to be a dishonorable profession by some".

APPENDICES

APPENDIX A

Core File Organization

This tutorial describes the contents of the core file (called core[1]) that is created when a process is interrupted by a program fault. Because the makeup of the core file is totally system dependent, the information contained herein is guaranteed to be accurate only for the UNIX-PC. However, the general concepts may be used to as a guide in investigating core files on other UNIX based systems.

The core file is used by debugging software to provide an image of the state of the CPU and user memory at the time a program fault occurred. The makeup of the core file is not directly documented. A diagram of the general layout of core is shown in Figure A.1. It was created by examining core with a tool similar to od[2] and applying structures to the output that can be found in the following header files:

```
/usr/include/sys/param.h
/usr/include/sys/pte.h
/usr/include/sys/reg.h
/usr/include/sys/user.h
```

The core file consists of a number of discrete sections as shown in Figure A.1. Each section is further identified by both its offset into the actual core file and its associated mapping into physical memory space.

1: CORE(4), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T

2: OD(1), in "AT&T UNIX System V User's Manual", Vol. II, (1985-1986), AT&T

This mapping derived through the use of the above header files. A brief description of each major section is as follows:

* User Blocks, 0 and 1: Contains all per-process data that doesn't need to be referenced while the process is swapped (Figure A.1). If the number of page table entries exceeds 512, a second page is allocated (User Block 1).
* .data: Contains an image of the contents of initialized variable space in user memory.
* .bss: Contains an image of the contents of uninitialized variable space in user memory.
* User Stack: Contains an image of the user stack space of user memory. A detailed examination of the stack can be found in Appendix D.

A detailed view of the User Block is shown in Figure A.2. Contained within the User Block is an image of the kernel stack area, the CPU registers, the user structure and the process entry table. The user structure is the "glue" that relates a host of process data (refer to sys/user.h for the exact makeup of the structure). It is located at relative offset 0x900 into the core file. Its members include pointers to the CPU registers (user.ar0), information about the header of the executable file (user.uarea_exdata), process user and group id information (user.u_uid, user.u_gid, etc.), to name a few.

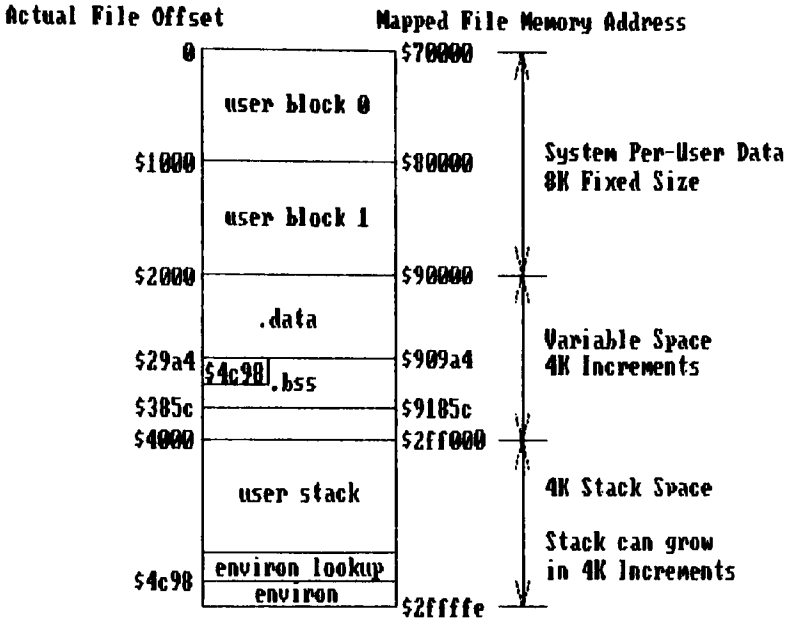

Figure A.1 Core File General Format

A debugger, such as SDB, utilizes this information to build tables, map offsets in files to physical memory and reference pointers to other files such as the executable file.

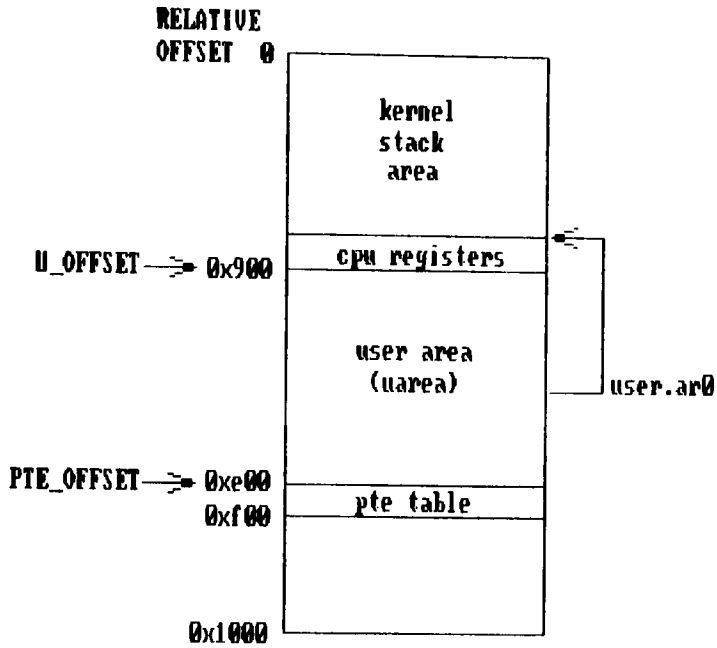

Figure A.2 User Block Format

A good example of the process of creating such tables is provided in the source code of SDB; specifically the setsym and setcor functions found in setup.c. This procedure is not trivial to follow but well worth understanding how the executable and core files are organized and interrelated. The state of the CPU registers are saved in the core file. The location of Address Register 0 (AR0) is pointed to by the user structure (user.ar0). The values of the remaining registers are simple offsets to the next seventeen words.

The incore page table entries (pte table) contains a set of structures that can be referenced to map the swap space for areas that are used or available for demand paging.

This concludes the discussion of the documented regions of the core file.

APPENDIX B

Interprocess Queue Message Types

Each interprocess queue message has a message type (mtype) tag associated with it when it is transmitted. This tag is used by GSD at the receiving node to identify what actions should be taken upon receipt of that message. The following table lists each base message type, direction of transmission, and the encoded meaning of the message.

MTYPE	DIRECTION	MEANING
1	Wx -> GSD	Message is pure text data
1	GSD -> Wx	Message is pure text data
2	W1 -> GSD	Adjust cursor position for mouse hit
2	GSD -> Wx	Clear window x
3	W1 -> GSD	What is the current source file name
3	GSD -> W1	Here is the current source file name
4	GSD -> W1	Print this line in reverse video
5	W1 -> GSD	Move +/- lines in the source buffer
5	GSD -> W1	Here is the cursor window position
6	Wx -> GSD	Here is the current window height
7	W1 -> GSD	How many lines in the source file
7	GSD -> W1	Here is the number of source lines
8	GSD -> Wx	Here is a message for the prompt line
9	W1 -> GSD	What is the current source line number
9	GSD -> W1	Here is the current source line number
10	GSD -> W2,3	Start of Block Transmission marker
11	GSD -> W2,3	End of Block Transmission marker

MTYPE	DIRECTION	MEANING
13	W3 -> GSD	Send the local variables
14	W3 -> GSD	Send the global variables
15	W3 -> GSD	What is the current procedure name
15	GSD -> W3	Here is the current procedure name
16	W3 -> GSD	Explode the structure in the message
95	GSD -> Wx	Initialize window to default settings
96	Wx -> GSD	Window x is now active (foreground)
97	Wx -> GSD	Select another window (x) to foreground
97	GSD -> Wx	Wakeup Window x to foreground
98	Wx -> GSD	Synchronize the debugger to Window x
99	Wx -> GSD	Initiate window shutdown broadcast
99	GSD -> Wx	Shutdown Window x

Abbreviation Key:

GSD: the underlying symbolic debugger process.

W1, W2, W3: Window 1, 2, or 3 child processes.

Wx : Wx, any window x of the three child processes.

APPENDIX C

Representation of Symbol Types in the Symbol Table

One of the most important sources of information that is referenced by SDB is the symbol table of the executable file that is being debugged. This table consists, in part, of symbolic representations of variables that were declared in the original source document and it subsequent conversion by the compiler into a form suitable for later use. The exact format of the symbol table is described in detail within the specifications of the Common Object File Format [1]. It is the type of the symbol and its use that will be the focus of this appendix.

The type field in the symbol table contains information about the fundamental (basic) and derived type(s) for the symbol. Each symbol has exactly one fundamental type but can have more than one derived type. The format of the 16-bit type entry is shown Figure C.1 .

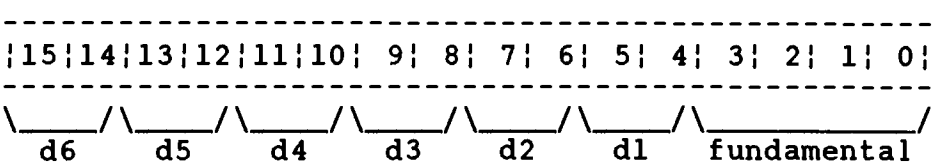

Figure C.1 Symbol Type Storage Format

Bits 0 through 3 represent the fundamental types given in Table C.1 . Bit pairs 4 through 15, marked "d1" through "d6", represent levels of the derived types given in Table C.2 .

1: Common Object File Format, in "AT&T UNIX System V Programmer's Guide", (1985-1986), Chap 18, pg 1-67, AT&T

The following examples demonstrate the interpretation of symbol table entries with respect to type:

```
int intvar;
```

Here intvar is the name of an integer. The fundamental type of intvar is 4 (integer) and the d1 field is 0 (no derived type). Therefore, the type word for intvar has a value of 0x4 indicating an "integer", as shown below:

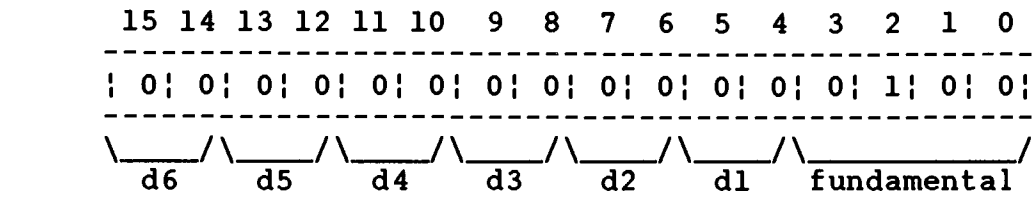

Figure C.2 Symbol Type Storage Format for intvar

```
char *charptr[3][7][9][2]
```

Here charptr is a 4-dimensional array of pointers to characters. The fundamental type of charptr is 2 (character), the d1, d2, d3 and d4 field each contain a 3 (array) and the d5 field is 1 (pointer).

Therefore, the type word for charptr has a value of 0x1ff2 indicating a "4-dimensional array or pointers to characters", as shown below:

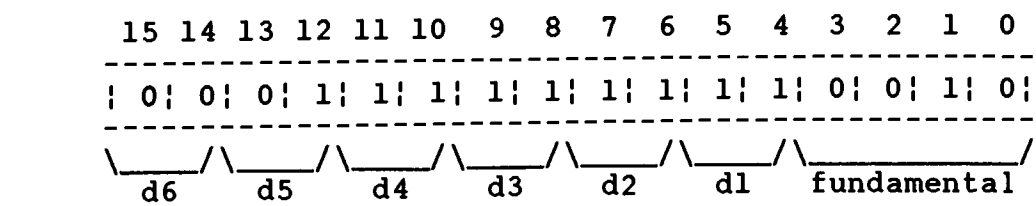

Figure C.3 Symbol Type Storage Format for charptr

Given the above encoding information, it is then possible to

develop a technique through which variables of derived type can be automatically expanded to yield its individual members. A simple method to do this has been implemented in the GSD debugger. Refer to Chapter 6.6, The Variables Window, for more information.

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Table C.1 Fundamental Types.

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

Table C.2 Derived Types.

APPENDIX D

68010 and System Stack Organization

This appendix will serve as a tutorial of the stack organization in memory on the UNIX-PC and how it functions with respect to the 68010 CPU. This knowledge is beneficial in understanding certain debugging operations such as stack traces. Many of the constants that are relevant to machine dependent features can be found in the following header files:

```
/usr/include/sys/param.h
/usr/include/sys/reg.h
/usr/include/sys/user.h.
```

The top of the stack is located at the physical address 0x300000 and grows downwards (more negative) to an initial physical limit of 0x2ff000. More stack region may be added by the operating system in 4 Kbyte increments, as needed. The manner in which the stack is manipulated is dictated by the 68010 architecture and by the way that a compiler and/or assembler treats the various constructs of a programming language. The stack is organized in discrete units called frames. Each frame represents certain fixed and variable pieces of information that map to each function.

Definitions

There are a several terms that should be defined before proceeding to an example:

* Program Counter (PC): A register of the PC that points to the next machine instruction to be executed.

* Stack Pointer (SP): A register of the CPU (AR7) that serves as a

pointer to the current location on the stack.

* Frame Pointer (FP): A register of the CPU (AR6) that serves as a pointer to the stack location of the previous frame pointer. This is in essence a link back up the stack that can be used to traverse it towards the top of the stack from the current SP location.

Function Calls

The following example demonstrates how the stack is manipulated when a function call is made:

* When a function call is made through a jump to subroutine (JSR) or branch to subroutine (BSR), the CPU pushes the PC (the return address) onto the stack. The PC is then set to the address of the new function.
* If a function call includes an arguments list, the arguments are pushed onto the stack in reverse order before the return address. If the argument is a global variable the contents of the actual address is pushed. If the argument is a local variable, the contents on the stack relative to the FP is pushed. If the argument is a pointer the memory address is pushed.
* The first machine instruction of the new function is always a link (LINK). LINK instruction carries two arguments. The first one specifies which register will serve as the FP and the second one specifies the amount of memory to be reserved for the local variable area. Link causes the contents of FP to be saved on the stack so that the address of the old FP is not destroyed. The SP is then advanced to the bottom of the

local variable reserved area and the FP is set equal to the SP to also point to the bottom of the reserved area.

* Now the stack are is free to be used for local stack storage operations or until another JSR, BSR or return from subroutine (RTS) instruction is encountered.

```
int statvar;

    first (in1, in2, in3)     /* FRAME 2 */
    int in1, in2, in3;
         {
         int var1;

         var1 = in1;
         second (input);
         }

    second (input)            /* FRAME 3 */
    int input[8];
         {
         input[1000] = 10;    /* point of crash */
         }

    main ()                   /* FRAME 1 */
         {
         int index;
         int listing[10];

         statvar = 43;
         first (statvar, index, 44);
         second (index);
         }
```

Figure D.1 Example Code Segment

* The end of a function is marked by two machine instructions; unlink (UNLK) and RTS. The UNLK instruction carries an argument to specify which CPU register is the FP. It causes the SP to be loaded with the contents of the FP register (the pointer to the previous contents of the FP). The contents of the location

that SP is pointing at is then loaded into the FP. The RTS instruction then sets the PC to the instruction following the JSR or BSR call to the returning function.

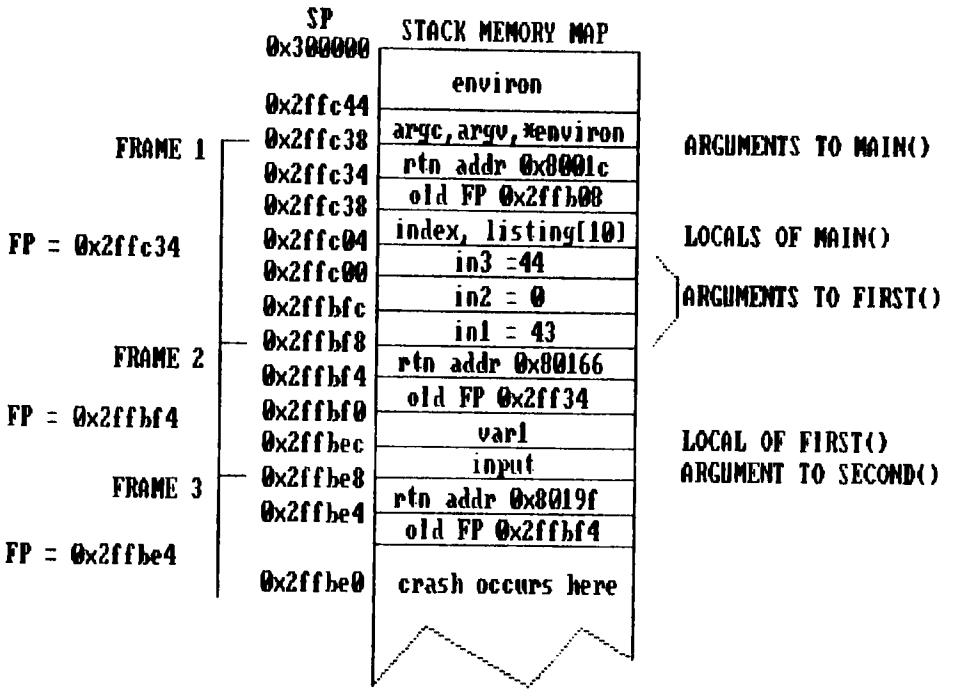

Figure D.2 The Basic Machine Stack Organization

Program Execution

In the following example a program is invoked by a keyboard command. It then proceeds to execute through three function calls before a core dump occurs. A snapshot of memory stack region, shown in Figure D.2, represents the program execution of the source code shown in Figure D.1. Its meaning is as follows:

* When an assembly language file is processed by the assembler, a section of code, referred to as the preamble is added. This preamble section contains start-up and shutdown information for the system including stack and frame alignment operations. Program execution always begins at address 0x80000.
* When a program is initiated the preamble section is executed. Through it the first item that is placed on the stack (beginning at 0x300000) is the environment list and its associated lookup table (see environ(5)). The pointer to the lookup table can be found in the first word following the .data section of memory. This area of memory is fixed during the life of a process and does not grow or shrink. Immediately following this region of memory is the dynamic stack area.

Now as execution proceeds, the following are pushed onto the stack.

FRAME 1

* The arguments to the function main (argc, *argv, and **environ) are pushed.
* The PC (0x8001C) to the underlying start-up code is pushed.
* The FP (0x2ffb08) to the calling frame is pushed. The FP is now set to 0x2ffc34 pointing to the previous FP on the stack.
* The SP is advanced around reserved memory (44 bytes) for the local variables of main.

* The values of the three arguments of the function call to first are pushed on the stack in reverse order.

FRAME 2

* The call to the function first causes the PC (0x80166) to be pushed.
* The FP (0x2ffc34) to the calling frame is pushed. The FP is now set to 0x2ffbf4 pointing to the previous FP on the stack.
* The SP is advanced around reserved memory (4 bytes) for the local variables of first.
* The value to the argument of the function call to second is pushed.

FRAME 3

* The call to the function second causes the PC (0x8019f) to be pushed.
* The FP (0x2ffbf4) to the calling frame is pushed. The FP is now set to 0x2ffbe0 pointing to the previous FP on the stack.
* The SP is advanced around reserved memory (32 bytes) for the local variables of second.
* The program crashes due to the array exceeding its memory boundary.

Had the prior example not crashed, the entire process would have reversed itself as the function calls were returned.

Using the Stack for Debugging Purposes

The core image of the stack area is an exact, preserved

representation of its state at the time of the program fault. It can be searched in several ways in order to extract different types of information that can be useful in determining the failure mode of the program in question.

The first debugging output is the stack trace. Because the stack is formed in the order of program execution, it can be traversed from the current stack pointer back to the top of the stack. Because the stack is organized into discrete frames, its is possible to identify specific areas of the frame such as: the frame pointer, the return address, variable and argument areas. The debugger can then cross reference data, such as the return address, against the symbol table to determine the function that the frame represents. This information then is formatted into a map that represents the order of program execution by function calls and the values of arguments (if any) that are associated with each function call.

The second debugger output is the display of the value of variables that are local to a particular function. Only functions that are currently on the stack (active functions) can have their local variables evaluated. This should be obvious as these variables do not have memory space made available until such time that a function call via the LINK instruction allocates it. Locating a variable that is local to a particular function requires a comprehensive stack trace in order to find all instances of it on the stack. Multiple instances of this would occur in cases such as recursion.

REFERENCES

1. T. Cargill, "Debugging C Programs with the Blit", AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, (October 1984), pg 1633-1647.
2. C.H. Small, "Software Debuggers Struggle to Meet Engineer's Needs", Software, Vol. 30, No. 10, (December 1985), pg 143-150.
3. M.S. Johnson, "A Software Debugging Glossary", ACM SIGSOFT/SIGPLAN, 1983 Symposium on High Level Debugging, Vol. 8, No. 4, (August 1983), pg 53-70.
4. J.D. Johnson and G.W. Kenney, "Implementation Issues for a Source Level Symbolic Debugger", ACM SIGPLAN, (March 1983), pg 149-151.
5. T. Cargill, "The Blit Debugger (Preliminary Draft)", ACM SIGSOFT/SIGPLAN, 1983 Symposium on High_level Languages, Vol. 8, No. 4, (August 1983), pg 190-200.
6. P. Mateli and G. Radack, "Integrating Data Structure Diagrams into Source Level Debuggers", ACM 14th Annual Computer Science Conf., 1986 Proceedings, (February 1986), pg 407.
7. M. Moran, "Toward Graphical Animated Debugging of Concurrent Programs in Ada", IEEE Computer Science Press, 8th International Conference on SE, (1985), pg 339-345.
8. E. Adams and S. Munchnick, "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations", Software-Prac. & Exper., Vol. 16, No. 7, (July 1986), pg 653-669.
9. N. Wirth, "MacMETH, A Fast Modula-2 Language System for the Apple Macintosh", User Manual, 2.0, Institut fur Informatik, ETH Zurich, Switzerland, (August 1986).
10. J. Hennessy, "Symbolic Debugging of Optimized Code", "ACM Transactions on Programming Languages and Systems", Vol. 4, No. 3, (July 1982), pg 323-334.