

Visual Interactive Debugger

TopSpeed Modula-2



TopSpeedTM Modula-2

For IBM[®] Personal Computers and Compatibles

Visual Interactive Debugger

Jensen & Partners International

LICENSE STATEMENT

Jensen & Partners International hereby grants you a non-exclusive license to use the TopSpeed Modula-2 Visual Interactive Debugger (VID). You may use the software on any computer, **provided that the software cannot possibly be used on more than one computer at the same time.** You may make copies of the software for the sole purpose of loading it into the computer on which you wish to use it, or for keeping a maximum of two backup copies.

You are furthermore granted the right to sell programs developed by you using the program modules provided in VID. You are not allowed to re-sell or give away any part of the VID source code.

WARRANTY

Jensen & Partners International warrants the enclosed diskette to be an accurate copy of the master disk, and will replace any defective copy free of charge for a period of 60 days after purchase.

Jensen & Partners International hereby explicitly disclaims all other warranties whether expressed or implied, including without limitation the implied warranties of merchantability, and fitness for any particular purpose. Jensen & Partners International shall have no liability for consequential, incidental, exemplary or special damages, including lost profits.

Jensen & Partners International reserves the right to change the product at any time, without prior notice.

TECHNICAL SUPPORT

To qualify for technical support, fill in and mail the enclosed registration card.

TopSpeed is a trademark of Jensen & Partners International.

TechKit is a trademark of Jensen & Partners International.

IBM is a registered trademark of the International Business Machines Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

Copyright © 1988 by Jensen & Partners International. All rights reserved.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

1	Introduction	1
	What VID Will and Won't Do for You	2
2	Tutorial	5
	Starting VID	5
	Introduction to VID Keys, Menus and Windows	6
	Giving Commands in VID	7
	Special Keys	7
	Command Help Key	7
	A Sample Debugging Session	9
	The Source Window	9
	The Execution Cursor	9
	Moving with the Execution Cursor	10
	The User Cursor	10
	Tracing Procedure Calls	11
	Setting and Moving to Breakpoints	12
	Examining Variables	12
	Setting Sticky Breakpoints	13
	Swapping Screens to View Program Output	13
	Clearing Breakpoints	14
	Setting Delayed Breakpoints	14
	Examining and Tracing Local Variables	15
	Returning to the Calling Procedure	15
	Skipping Procedure Tracing	16
	Setting Watch Variables	16
	Moving Windows	16
	Looking at a Specific Procedure	17
	Finding Specified Strings	17
	Returning to the Execution Cursor	17
	Disabling and Enabling Modules from Debugging	17
	Tracing Continuous Program Execution	18
	Tips for Exploring on Your Own	18
3	Reference	19

Installing VID	19
Customizing Your Debugger Environment	19
Installing VID in the TopSpeed Environment	22
Compiling Modules to Debug	22
Linking Your Program	23
Running VID	23
Running VID from within the TopSpeed Modula-2 Environment	23
Expressions in VID	24
VID Command Line Options	26
The Source Window	28
Commands for Moving in the Text	28
Commands for Moving in the Running Program	29
Commands for Handling Windows and Screens	30
Commands for Exiting VID	31
Additional Commands for Moving in the Source File	31
Highlighting and Marking Lines	32
VID's Menu Structure	33
VID's Main Menu	35
The Breakpoint Commands	36
Introduction to Breakpoints	36
The Breakpoint Menu	37
Set Breakpoint ([+] or [B][P])	37
Clear Breakpoint ([C] or [B][C])	38
Set Sticky Breakpoint ([*] or [B][S])	38
Set Delayed Breakpoint ([B][D])	38
Set Delayed/Sticky Breakpoint ([B][K])	38
Set Expression Breakpoint ([B][E])	38
Monitor Breakpoint ([B][M])	39
Set User Breakpoint ([B][U])	39
Clear All Breakpoints ([B][A])	39
The Examine Commands	40
Examine Menu	43
Examine Variable ([?] or [E][V])	43
Global Variables ([Alt][F3] or [E][G])	44
Procedure Parameters ([Alt][F4] or [E][P])	44
Local Variables ([Alt][F5] or [E][L])	44
Disassemble Statement ([E][D])	44
Evaluate Expression ([E][E])	44
Stack Words ([E][S])	44
Memory Bytes ([E][B])	44
Memory Words ([E][W])	45
Memory ASCII ([E][A])	45
Registers ([E][R])	45
Floating Point Stack ([E][F])	45
The Go Commands	45

The Go Menu	45
Go until Breakpoint (G B)	46
Go Trace Source (G T)	46
Go to User Cursor (G C)	46
Go until Return from Current Procedure (G R)	47
Go until Procedure Call (G P)	47
Go until Expression True (G E)	47
Restart the Program (G S)	47
Go Forever (G X)	48
The Trace Windows	48
The Trace Menu	48
Procedure Calls (F4 or T P)	49
Active Variables (F5 or T V)	50
Registers Trace (T R)	51
Assembler Trace (T A)	52
Watch Variables	53
Watch Variable Trace (T W)	55
Set Watch Variable (T S)	56
Delete Watch Variable (T D)	57
Commands for the Source Window	57
The Source Menu	58
Change Module(F3 or S C)	58
Enable Module (S E)	58
Disable Module (S D)	58
Previous Procedure (F7 or S P)	58
Next Procedure (F8 or S N)	59
Locate Procedure (F9 or S L)	59
Find String (Ctrl Q F or S F)	59
Goto Line (S G)	59
View File (S V)	59
User Options	60
The Options Menu	60
Hex Output (O H)	61
Screen Swap (O R)	61
Volatile Check (O V)	61
Highlight Code (O C)	62
Status Line (O L)	62
Single Process (O P)	62
Trace Assembler (O T)	62
Trace Delay (O D)	62
Save Options (O S)	62
Auto Save Options (O A)	62
Default Options	63
Run-Time Errors	63

4	Advanced Debugging Topics	65
	VidIf: The Program Interface to VID	65
	Tracing from Your Program	65
	Display Output Control	68
	Returning Control to VID	70
	User Breakpoints	70
	Debugging Multiple Processes	71
	Debugging a Multiple-Process Program	72
	Debugging Multiple Processes One at a Time	73
	Modula-2 Utilities	73
	M2DASM	73
	M2PROF	76
	M2SIZE	78

Chapter 1

Introduction

Welcome! The TopSpeed Visual Interactive Debugger (VID) provides an easy-to-use, but powerful, debugging environment. You can use VID to trace your program's execution — to find errors or simply to study the workings of a program or procedure.

This manual shows you how to install and use VID. In addition, you'll find a summary of the available commands for VID, and also a chapter on Advanced Debugging Techniques. We'll assume you're comfortable using Modula-2, and that you've got the TopSpeed Modula-2 system installed — so that you can do the compilation and linking suggested in later sections.

The interface for VID is designed to be powerful but straightforward. Debugging can now be fun as VID brings your source code to life!

In addition to a powerful debugger, your VID package includes an installation program that lets you customize your VID environment, and a module for letting your program output information in a VID trace window. The package also contains three utilities: for disassembling your program code file, for watching your program's execution and providing a profile of execution patterns, and a program to tell you the size of procedures and modules.

What VID Will and Won't Do for You

VID won't find the bugs in your program by itself. However, the debugger *will* give you a high-level view of the program's execution, and will make it easy for you to get to the heart of the bug quickly.

VID won't tell you what's causing an error in your program. However, VID does give you lots of different ways to look at your program's variables and values. This makes it easier for *you* to determine what might be wrong with your program.

VID has many powerful and attractive features, and lets you do a great variety of things when debugging your programs. Some of the things you can do are summarized in the following list.

- You can do a high-speed trace of your program's execution, or you can slow the execution down to enable more careful observation. You can set traps at which program execution will pause to give you an opportunity to examine the state of the program (values of variables, current stack of function calls, etc).
- You can execute the entire program or individual procedures step-by-step (single step) to let you study the values of variables at each step.
- You can trace the value of a particular variable or calls to a procedure through the source code, as your program executes. You can set the debugger to automatically show the values of any variables that are currently active in the executing program.
- You can specify certain statements in the program at which the execution will stop until you have examined what's of interest to you. Such selected statements are called *breakpoints*. VID lets you set up to 100 or so breakpoints in a program.
- You can view, modify, and trace variables in full symbolic form. That is, you can refer to the variable by name, rather than having to specify a particular address explicitly.
- You can specify expressions for values, and you can use the standard operators as well as active variables from the program.
- You can trace the sequence of procedure calls. When this feature is on, you can tell at any point in the program what procedure is executing, and you get a list of all the procedures currently suspended because of calls to other procedures. The trace window that contains this information also tells you the source line at which each of the procedures was suspended.
- You can dump the entire screen or the active window to a disk file for later examination.
- You can cycle through each of the windows you have open. By making each window active in turn, you can keep track of all aspects of the program's execution.

- You can trace your program at the level of assembler code, and can even trace the contents of the machine registers during program execution.
- You can modify both high-level and low-level program elements. In addition to being able to modify high-level variables, you can modify registers, memory, and both the program stack and the 8087 stack.
- You can disassemble the program, and can get mixed source and assembler display.
- You can set various kinds of breakpoints — including “one-shot” breakpoints, where the program will stop once, and “sticky” breakpoints, where the program will stop each time it reaches the statement during execution.
- You can watch variables and expressions — so that their values are displayed continuously during program execution and updated when they change in the program.
- Before running your program, you can initialize program memory to any byte value you specify. In this way, you can find errors that might occur because of uninitialized variables.
- VID includes a powerful collection of commands for moving around in your program. For example, you can go to particular statements, to procedures, to statements after returning from procedures, to breakpoints, until an expression becomes true, and so forth. These commands make it easy to move to whatever part of your program needs the most careful examination at any given time.
- You can view the contents of text files other than source modules. For example, you might need to consult the documentation or definition files for a program you're debugging. VID makes this possible, even if the documentation file is larger than available memory.
- You can tell VID to save the program's state at the end of your session, so you can continue the debugging session from exactly this point after you've had a break for coffee or taken the time to study the program by other means.
- The debugger lets you control how to handle run-time errors. If such an error occurs in your program, VID takes you to the statement at which the error occurred. You can then specify whether to pause or continue program execution.
- The debugger is very easy to use. You can select commands from menus, or you can specify them using shortcut keys.

In the next chapter, you'll find a brief tutorial that lets you go through a debugging session, and that introduces you to some of these features. You can explore some of the commands and can become more comfortable with VID's interface.

You'll find that the debugger's command structure is very similar to the structure you're used to from TopSpeed Modula-2. You'll find pop-up menus, which you can activate — generally with a single keystroke. However, you can also bypass these

menus by using shortcut command keys — generally one or two keystrokes. So let's get started!

Chapter 2

Tutorial

To show you how easy it is to debug a program let's go through a debugging session. We'll debug the program **VidDemo** supplied with VID. If you're impatient to start using VID to debug your own code then see the sections on compiling and linking modules for debugging. These sections are in Chapter 3, along with summaries of the commands available for VID.

Starting VID

VID needs a number of files from which it extracts the information it uses to run a debugging session. These files are:

1. Source (**.MOD**) files for each module being debugged
2. Debug (**.DBD**) information files for each module being debugged
3. A map (**.MAP**) file for the program
4. The program (**.EXE**) file itself

To debug the test program you need the following files:

```
VidDemo .MOD   VidDemo .DBD
PushOver.MOD   PushOver.DBD
VidDemo .MAP   VidDemo .EXE
```


These files are included in your VID package in the subdirectory **Examples**. They are in the form the debugger wants. This saves you the trouble of compiling and linking the program's files.

First, copy these files to the directory in which you want to start trying out VID. You should also make sure that the files **VID.EXE** and **VID.OVL** are in the same directory or are reachable through your **PATH**.

Once you've got these files arranged the way you want, you can run VID by typing the simple command

```
vid viddemo
```

This will immediately start VID up and running. The first thing VID will do at this stage is to create a file, **VidDemo.DB\$**. This is a temporary file, and is used by VID to speed up the debugging session.

The file contains (in compressed form) the **MAP** and **DBD** information as well as some information about the **.MOD** files. For speed, the **DB\$** file will be remade only when the map file changes. The **DB\$** file is useful for the debugger, but is not necessary. You can delete it at any time if you wish, as it can always be recreated — a process that takes only a second or so.

Once VID has finish creating this file, you'll see the main VID screen, with the introductory banner in the center of the screen. The source for **VidDemo** is already on display in the background.

To remove the banner, press the **Escape** key (or give any VID command). You are now ready to start debugging **VidDemo** (easy wasn't it)!

Before we start the sample session, let's look briefly at VID's design and at some of its general features.

Introduction to VID Keys, Menus and Windows

VID always tries to provide as much information as you need on the screen without cluttering the screen. To avoid such clutter, certain information is displayed only if you request it.

To accomplish this, VID has multiple overlapping windows, each of which contains a specific type of information. You can open and close these windows as needed, and you can move them around at any time.

Giving Commands in VID

You can give commands to VID by one of two mechanisms:

1. You can use the pop-up menu system. To invoke the main menu for this, press **F10**. This menu system will be familiar from the TopSpeed environment. Browse around the menu, and take a look at all the commands VID provides for you. To find out more details about a particular command you can consult Chapter 3.
2. You can use “shortcut” key sequences. These have the same functionality as the menu commands but are quicker to type. You’ll probably use the menu-system when you first start using VID. As you become more familiar with VID, you’ll probably find yourself using shortcuts more and more. As in the TopSpeed environment, a menu will pop up if you type just part of a shortcut sequence.

Special Keys

In addition to the menu shortcut keys, certain other actions have been assigned single keys. These include actions (such as setting breakpoints) that are performed frequently when debugging. The bottom two lines of the VID screen contain a small help window that briefly lists some of these keys.

This help window will generally show which keys are applicable to the particular “mode” you are in. When you become more experienced with the debugger’s commands, you may find it useful to remove these help lines — to give yourself more room for other information.

Command Help Key

To get a list of special keys and top-level menu commands, press the **F1** key. This puts up a “Key help” reference list, as shown in Figure 2-1.


```

File: VidDemo                                     line: 92
WhoPlaysFirst : PushOver.Player ;

VID Keys
┌──────────┴──────────┐
│ FUNCTION KEYS      │ COMMANDS      │ MENUS      │
├──────────┴──────────┘
F1 Key help          space Single step      F10 Main menu
F2 Registers          P Skip procedure call  B Breakpoint
F3 Change module     + Set breakpoint       C Goto
F4 Procedure calls   - Clear breakpoint     E Examine
F5 Active variables * Set sticky breakpoint S Source
F6 Cycle windows    ? Examine variable     T Trace
F7 Previous procedure
F8 Next procedure
F9 Find procedure
F10 Main menu        Esc Close top window
AltF2 Assembler trace Alt-X Exit from VID
AltF3 Global variables
AltF4 Procedure parameters
AltF5 Local variables

ScrollLock toggles window control mode
CtrlBreak pauses an executing program

IF exit THEN
VID Keys: Space-Step P-Skip Screen swap Esc-Close window Q-Quit VID
V1.00 Menus: Breakpoint Examine Goto Source Trace Options F1-Key Help F10-Main

```

2.1. VID's Key Help list.

You can press the **Escape** key to exit from this list (or from any other VID window), or you can press one of the keys listed — to execute that command.

A Sample Debugging Session

The Source Window

Let's get back to the debugging session. After you get rid of the product banner, your screen will look like the one in the following figure.

```

File: VidDemo                                     line: 92
WhoPlaysFirst : PushOver.Player ;

BEGIN (* VidDemo program *)

WhoPlaysFirst := PushOver.machine ;

OpenIntroWindow ;          (* display text introduction *)

WrText(IntroText) ;

PushOver.Init ;             (* initialize pushover variables *)
PushOver.DrawBoard ;        (* and draw the empty board *)

k := IO.RdKey() ;           (* wait for user to finish reading *)

Window.Close(IW) ;          (* close introduction window *)

LOOP

PushOver.PlayGame(WhoPlaysFirst, exit) ;

IF exit THEN
VID  Keys: Space-Step P-Skip  Screen swap Esc-Close window Q-Quit VID
01.00 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main

```

2.2. VID opening screen.

VID automatically starts running the program being debugged, and will pause at the first statement to be executed in the modules being debugged. Above the help lines on the screen, you'll see some Modula-2 source code. Above that, on the top line, you'll see a file name and line number. The file refers to the source on display and the line number refers to the highlighted line across the source text. (This line is blue on color monitors.)

The Execution Cursor

This highlighted line shows the position in the source text that is about to be executed. This line is known as the *execution cursor*. The execution cursor will move as you single step or execute through your program.

The execution cursor highlights a statement *before* it has executed. To execute a single statement, just press **[Space]**. The statement will be executed, and you will see the execution cursor move to the statement that will be executed next.

After you press **[Space]** in **VidDemo**, the execution cursor indicates that the next instruction will be a procedure call, to **OpenIntroWindow**.

Moving with the Execution Cursor

Let's see what happens when the execution cursor is on a procedure call. When the call to **OpenIntroWindow** is highlighted, this means that the call will be the *next* instruction carried out. If you press **[Space]** again, you'll find that the execution cursor is now sitting on the first executable statement in procedure **OpenIntroWindow**.

When control is transferred to another procedure through a call, **VID** continues the single-step execution at the start of that procedure. The start, in this context, refers to the first line of executable code — that is, code other than the procedure heading and variable declarations. Once the procedure finishes executing, the execution cursor will return to the next statement to execute in the calling procedure or in the main program.

Thus, the execution cursor moves along the same paths as the program does when it executes. "Next" for the execution cursor means the next statement the *program* will come to, *not* the next statement in the source listing.

The User Cursor

You can also move through your source file independently of the execution cursor — by using the normal PC cursor and page movement keys. If you try pressing the **[↓]** key (for example), you will see another line on your screen. This line is red on a color monitor, and underlined on a monochrome monitor.

This movable cursor is called the *user cursor*, and it moves independently of program execution. You can use this cursor to point out to **VID** the places where you want something to happen — for example, a breakpoint to be set.

Move the user cursor down until it's on the following line.

```
FrameOn      := TRUE ;                (* frame enabled *)
```

You can continue executing the program from this instruction, if you wish. To do this, move the execution cursor to the user cursor position:

- Press **[F10]** for the main **VID** menu
- Press **[G]** to select the **G**o menu
- Press **[C]** to go to the current (user) **C**ursor location

You'll find that the execution cursor is now on the same line as the user cursor.

Now move the user cursor again, to the last line of **OpenIntroWindow**:

```
IW := Window.Open(WD);                (* open the window *)
```

To move the execution cursor to this line, just press **[G]** and then **[C]**. This pair of keys has the same effect as the menu selections, but is much quicker to specify and to execute.

Press **[Space]**. This time the execution path does not continue with the first statement in **Window.Open**. This procedure is skipped, because it's in a module that is not being debugged.

At this point, the execution cursor should be highlighting the following line:

```
END OpenIntroWindow ;
```

Press **[Space]** again. Notice that the execution cursor is now back in the main program, at the statement after the call to **OpenIntroWindow**. This statement is a call to procedure **WrText**. If you press **[Space]** again, the execution cursor will move to the first line of **WrText**.

Tracing Procedure Calls

VID's single-step facility lets you execute your program one statement at a time. This allows you to observe the sequence of individual instructions as your program executes.

You can also trace the sequence of procedure calls, and can determine what procedures are currently active — that is, either executing or called but temporarily suspended while another procedure executes. To do this, press **[T]**. In a second or so, the Trace menu will come up. Press **[P]** to select the procedure trace option. After this, your screen will look like the one in Figure 2-3.

File: VidDemo		line: 18
VAR		Active Procedure Stack—Line
line	: ARRAY[0..77] OF CHAR ;	VidDemo. 98
lineno	: CARDINAL ;	VidDemo.WrText
BEGIN		
lineno := 0 ;		
LOOP		
Str.ItemS(line,text,' ',lineno) ;		
INC(lineno) ;		
IF line[0]=0C THEN		
EXIT ;		
END ;		
IO.WrStr(line);		
IO.WrLn;		
END ;		
END WrText ;		
PROCEDURE OpenIntroWindow ;		
VAR		
WD : Window,WinDef ;		
BEGIN		
WITH WD DO		

VID	Keys: Space-Step	P-Skip	Screen swap	Esc-Close window	Q-Quit VID
U1.00	Menus: Breakpoint	Examine	Go Source Trace	Options F1-Key Help	F10-Main

2.3. Procedure Trace window.

This window displays the procedures currently active. All but one of these is suspended temporarily, and only the last procedure will be executing. For each suspended procedure, the window includes the source line on which the procedure was suspended — because of a call to the next procedure on the list.

In Figure 2-3, you can see that the main program (**VidDemo.**) was suspended on source line 98, at which point procedure **WrText** from the **VidDemo** module started executing. The current procedure is highlighted in this window. You can leave this window on the screen as you trace your program execution. The window will be updated with each new procedure call.

If you want to turn procedure tracing off, just press **[Escape]**. This will close the trace window, and will turn procedure tracing off.

Setting and Moving to Breakpoints

In **WrText**, move the *user* cursor to the line

```
INC(lineno) ;
```

Press **[+]** to set a *one-shot* breakpoint. (You can also accomplish this by pressing **[B]** for the breakpoint menu, and then pressing **[P]** for a one-shot breakpoint.) A special symbol, ►, appears to the left of the line at which the breakpoint has been set.

Very briefly, a breakpoint is a trap that makes program execution pause when it reaches the trap. This gives you the opportunity to examine the values of variables at this point in the program.

After you've set this breakpoint, move the user cursor down a few more lines, then press **[G] [B]**. This moves the execution cursor to the line at which a breakpoint had been set. (You could also have selected this command from menus — pressing **[G]**, waiting for the menu, then selecting **[B]** from the resulting menu).

Examining Variables

You can examine the value of a variable during program execution. For example, to see the current value of **lineno**, press **[E] [V]** (for **E**xamine **V**ariables).

A window will appear in the upper left portion of your screen. You'll be prompted for a variable name. Type **lineno** and press **[Enter]**. You'll find the variable has a value of 0.

At the top of the window you'll find the procedure in which the variable is defined (**WrText**). The bottom of the window shows the variable's address.

You can change the value currently stored there, simply by typing a new value, then pressing **Enter**. Try changing the value of **lineno** to 37.

To close the Examine window, press **[Escape]**.

Recall that the execution cursor highlights a line *before* it has been executed. Press **[Space]** to single step to the next line in the procedure. This executes the


```
INC(lineno) ;
```

line.

With the execution cursor on the next line, let's take another look at the value of `lineno`. You can open the Examine window quickly, by simply pressing **[F2]**. The last variable you examined will be the default one to examine this time.

Just press **[Enter]** to check this value. You'll find that the new value for `lineno` is 38 — an increment of 1 from the 37 you assigned the variable.

You can also examine the value of string variables (such as `line`). Open the Examine window (**[F2]** or **[E][V]** will do it). Enter `line` as the variable's name. You'll find a multiline display of each cell's value in the array. Again, you can change values if you wish.

Incidentally, notice that the marker (**[▶]**) indicating a breakpoint disappears as soon as the execution cursor moves on. This is because the breakpoint was set as a one-shot. Such breakpoints disappear after program execution has paused at the breakpoint.

Setting Sticky Breakpoints

You've seen how to set single-shot breakpoints when debugging. You can also set breakpoints that will remain, after you have executed through them. These are known as *sticky* breakpoints.

Move the user cursor to the line

```
IO.WrLn;
```

Then type **[B][S]** for (**B**reakpoint **S**ticky). You'll see a diamond-shaped character (**[◆]**) to the left of the statement. You can move the execution cursor to this line by pressing **[G][B]**, as in the previous example.

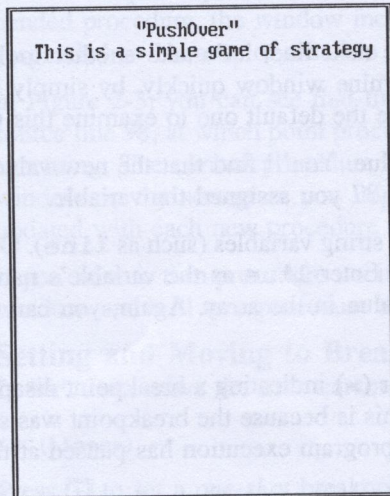
If you press **[Space]**, you'll see your screen flash quickly, as if something were being displayed. Notice that the breakpoint marker has not disappeared. This is because the trap remains when you set a sticky breakpoint.

You can execute the breakpoint line again by pressing **[G][B]** to move the execution cursor to the breakpoint again, then pressing **[Space]**. You'll again see your screen flash quickly.

Swapping Screens to View Program Output

Each of the calls to `IO.WrLn` writes another line of introduction to the screen. Thus, after executing the statement twice, two lines should have been written.

You can check this by looking at the program's screen — that is the screen that would be displayed if the program were executing normally. To see this screen, press **[F7]**. Your screen will look like the one in Figure 2-4.



2.4. Program screen after two lines have been written.

To get back to the trace screen, press **F7** again.

Clearing Breakpoints

You can clear a sticky (or any other kind of) breakpoint by typing **B C** — for **B**reakpoint **C**lear. The user cursor *must* be on the statement whose breakpoint you want to clear.

Your command will remove the trap set at the current statement. Program tracing will proceed normally through the statement.

Setting Delayed Breakpoints

You can also set *delayed* breakpoints, in which you specify that a trap is to be set, but only after the statement has executed a specified number of times. Thus your program's execution will only stop at the statement after a specified number of "quiet" traces.

To set such a breakpoint, use **B D**. For example, after you've cleared the breakpoints in the example, move the user cursor to the same line again

```
10.WrLn;
```

This time, set a delayed breakpoint — by pressing **B D**. You'll be asked to provide a delay. Enter, for example, 5. You'll find 5► to the left of the entry.

Then type **[G][B]** to move the execution cursor to the breakpoint. Press **[Space]** to execute the statement — that is, to write a line. This loop will execute 5 times (in the example).

After these iterations through the loop, the trap will be enforced, and program execution will pause at the line. Switch to the program screen **[V]**, where you'll see that 5 lines have been written since the last time you were shown the screen.

Notice that the breakpoint trap disappears after the single step stops at the breakpoint. This delayed breakpoint was also one-shot. You can also set a *delayed sticky* breakpoint, which will remain, and which will trap program execution every time the statement is executed the specified number of times.

Examining and Tracing Local Variables

Let's examine the values of the local variables while **IO.WrLn** is displaying the introduction. After seeing the five lines displayed by the program so far, the execution cursor on the trace screen should still be on the line whose breakpoint has just been removed.

To see the values of local variables, press **[E][L]**. This will open a window containing the values of variables local to the **WrText** procedure.

You'll find two variables listed in the window: **lineno**, with a value of 5; and **line**, which contains the string just written to the screen. Press **[Escape]** to close the Examine window.

Press **[T][V]** to trace the values of the local variables while you single step through the program. This opens a trace window, which remains on the screen until you press **[Escape]** to close it. You can continue stepping through your program with this window open, and you can watch as the values of **line** and **lineno** change as the program works through the **LOOP**.

Returning to the Calling Procedure

You've seen enough of **WrText** to learn something about the **VID** commands. Let's continue with the main program's execution. Press **[G][R]** — for **Go Return**.

This will move you to the statement after the call to **WrText** — that is, the statement to which control would return when **WrText** finishes executing. This statement is a call to a procedure in a different module:

```
PushOver.Init ; (* initialize pushover variables *)
```

Press **[Space]** to single step into this procedure. Notice that **VID** is now tracing execution in a different module — **PushOver**, as indicated in the top line of the Source window.

The Trace window for variables should still be open. As you single step through **PushOver.Init**, you'll see new variables added to the trace window.

To close the variable trace window, press **Escape**. This command will always close the active window — the one with a double frame.

Skipping Procedure Tracing

Suppose you're single stepping through your program, and you come to a procedure that takes a very long time to execute. You can skip over this procedure in your step-by-step execution.

For example, let's skip over a procedure in **VidDemo**. First, single step through **PushOver.Init**, until the execution cursor is back in **VidDemo** and the following line is highlighted:

```
PushOver.DrawBoard ;           (* and draw the empty board *)
```

When the execution cursor is on the procedure call — that is, when the program is just *before* the procedure call — press **P** instead of **Space**. The execution cursor will move immediately to the next statement.

If you skip over a procedure call, the procedure is still executed. It's just that the execution is not traced, unless there is a trap set in the procedure. In that case, execution will still pause at the trap — even if you're skipping the rest of the procedure.

Setting Watch Variables

Before assigning **IO.RdKey()** to **k**, set some variables to watch. Press **T** (for trace) and **S** (for set watch variable).

Enter **WhoPlaysFirst** in response to the prompt, then press **Enter**. You'll find a Watched Variables window on your screen, containing the current value of this variable.

Press **T S** again, and specify **k** as the second variable to watch. This variable will be added to the window.

Moving Windows

Press **Scroll Lock**. You'll notice that the help lines at the bottom of the screen have changed. You're now in a mode in which you can move windows around on the screen.

For example, to move the Watched Variables window down toward the bottom of the screen, press **↓** until the window is at the desired level. To move the window left and right, press **←** and **→**, respectively.

When you've got the window in the location you want, just press **Scroll Lock** again. This will put you back in tracing mode.

Now press **Space** to single step over the assignment statement — thereby executing it. The complete introduction screen will appear on the screen, with a prompt for you to press a key. After you press a key, the tracing screen will reappear. The value of **k** will have changed — depending on the key you pressed.

Looking at a Specific Procedure

VID makes it easy for you to trace the execution of whatever procedure you wish. For example, let's look at a procedure in the **PushOver** module.

Type **[S][L]** — for **Source Locate** procedure. In response to the prompt, type

PushOver.*

A window will appear, showing all the procedures in module **PushOver**. Select **PushOver.PlayGame** by moving the cursor to the selection, and pressing **[Enter]**. The user cursor will be at the beginning of this procedure in the **PushOver** module.

Finding Specified Strings

You can also find specific strings in your source. Type **[S][F]** — for **Source Find** string. Specify **finish** as the string to find, then press **[Enter]**.

The user cursor will move to the first occurrence of this token. Press **[Ctrl][L]** to repeat this search for the next occurrence of the specified string. Instead of **[S][F]**, you can also use **[Ctrl][Q][F]**.

Returning to the Execution Cursor

If you want to get back to the execution cursor after you've moved the user cursor away, you can always press the "go-home" key, which is **[Ctrl][Home]**.

This command will always put the two cursors at the same line, even if they are in different modules. For example, in the present context, this should put the cursor back in the **VidDemo** module — unless you did some single stepping through **PushOver.PlayGame**.

Disabling and Enabling Modules from Debugging

Sometimes you may want to speed things up a little, by concentrating on one module at a time. At such times it's convenient to be able to limit the debugging to that module.

You can disable modules from being debugged. For example, type **[S][D]** for **Source Disable**. Press **[Enter]** to select **PushOver** in the Select Module window.

This module is now disabled. The module's procedures will still be executed when called. However, if you single step through the program and you come to a call to a procedure in this module, the debugger will not trace through the procedure's execution.

You can re-enable a module by pressing **[S][E]**, and selecting the module you want to re-enable. For example, select **PushOver** to re-enable that module.

Tracing Continuous Program Execution

You can let the program execute on its own, while tracing execution. This saves you the work of pressing **[Space]** to move the program forward. Press **[G T]** (for **Go Trace**) to start executing the program with continuous tracing.

You'll find the debugger moves through the source code very quickly as it traces program execution. You can stop this tracing by pressing **[Ctrl Break]** at any time while the program is *not* waiting for input.

Let the program execute a bit, then press **[Ctrl Break]**. This will stop the continuous tracing. You can single step now, if you wish. You can also return to continuous tracing by pressing **[T C]** again.

Before doing that, let's slow down the rate at which the debugger traces through the program. Press **[O]** for the Options menu. One of the selections, **[D]**, lets you delay the tracing for a specified amount of time.

Select this option, then specify 50 in the resulting window. This will delay the trace by $\frac{50}{100}$ ths of a second, or 0.5 second. Press **[T C]** to return to continuous tracing. You'll find that the pace is now much easier to follow.

Tips for Exploring on Your Own

Continue this sample session on your own, to learn more about VID's features and commands. Although we've been using shortcut keys to give commands, remember that all commands are accessible through menus.

You can always get the main menu by pressing **[F10]**. You can get any submenu by selecting it from this menu or by specifying the first letter of the submenu's name. For example, pressing **[B]** and waiting will get you the Breakpoint menu. These submenus will, in turn, give you access to the available commands.

Chapter 3

Reference

Installing VID

You can install VID very easily. Just copy the **VID.EXE** and the **VID.OVL** files to wherever you wish to put the debugger. Once installed, you can run VID from any directory, provided the directory containing the **EXE** and **OVL** files is on your path.

VID can find files in other directories, provided it has access to an **M2.RED** file with information about these other directories.

Customizing Your Debugger Environment

The **vidinst** program included in your VID package lets you customize your VID working environment by installing window colors and specifying various other settings and options.

To start the program, type

```
vidinst
```

Your screen will look like the one in Figure 3-1.

VID installation	
A - Color window recolor	
B - Mono window recolor	
C - Screen swap default	
D - Black/White override	OFF
E - Snow check	OFF
F - Fill value	000
E - EMS no of 16K pages	-
G - Save setup	

Select option to change: **↑** move **↓** select **Esc** finished installation

3.1. Main menu for VidInst program.

As you can see, you can change various features of your working environment, including the background and foreground colors of the various VID windows, settings related to screen swaps, and so forth.

To select an entry from this menu, use the cursor keys to highlight the entry you want, then press **Enter**. For example, to change the color setting for a monochrome monitor, select the second entry.

You'll get a list of windows whose settings you can modify. The selection process from this menu is the same as for the main menu: use the cursor keys to highlight the window you want, then press **Enter**. After selecting, for example, the Disassembly window, your screen will look like the one in Figure 3-2.

Select Window
A - Source
B - Source Heading
C - Procedure Calls
D - Variable Display
E - Variables Trace
F - Watch Variables
G - Register Trace
H - Assembler Trace
I - Disassembly
J - User Trace
K - List Select
L - Error
M - Dialog
N - Prompt
O - Main Menu
P - Sub Menus
Q - Status Line
R - Keys Help

Statement Disassembly
Line 37 X2 := 37 ; V2 := 24 ;
57A9:002F MOV Word Ptr(BP-17),25
57A9:0034 MOV Word Ptr(BP-15),18
Line 39 Foreground := Window.White ;
57A9:0039 MOV Byte Ptr(BP-13),0F
Line 40 Background := Window.Red ;
57A9:003D MOV Byte Ptr(BP-12),4
Line 42 CursorOn := FALSE ;
57A9:0041 MOV Byte Ptr(BP-11),0
Line 43 WrapOn := FALSE ;
57A9:0045 MOV Byte Ptr(BP-10),0
Line 44 Hidden := FALSE ;
57A9:0049 MOV Byte Ptr(BP-0F),0
Line 46 FrameOn := TRUE ;
57A9:004D MOV Byte Ptr(BP-0E),1
Line 47 FrameDef := Window.DoubleFrame

Recolor Window: ←background →foreground PgUp/PgDn-select area Esc-exit

3.2. VidInst screen after selecting Disassembly window.

Use the  and  keys to cycle through the available foreground “colors,” and use the  and  to change the background. When the window looks the way you want it to, press  to return to the menu.

The screen swap defaults refer to the way in which VID will handle the process of switching between the debugging screen and the screen as it looks when the program is executing. The possible settings are summarized later in this chapter. The BW Override feature lets you force VID to use monochrome mode. The Snow Check lets you turn on snow suppression on CGA monitors that flicker. The Fill Value lets you specify the initial value to be stored in memory before the program being debugged starts executing. Finally, the EMS value allows you to specify whether you wish to use expanded memory for VID, and how many 16K pages should be allocated.

When you’ve finished setting all the features and colors you want, select entry G on the main VidInst menu, to save the settings. The values you specified will go into effect, and will be used each time you run VID.

You can also specify all but the window color settings on the command line when you invoke VID, as described later in this chapter.

Installing VID in the TopSpeed Environment

You can install VID as a function executable from the TopSpeed Modula-2 environment. In this case, VID will appear in the TopSpeed menu just like the linker and compiler, and possibly also the TopSpeed assembler, if you have the TechKit.

To install VID in the environment, add the following line to **M2.MNU**:

```
{V}id | 'evid %M' <AltV>
```

This new line should be either in the top-level menu or in a submenu (such as the line after **{I}nfo**).

To make the change, you can use either of the following methods:

- If you use the standard **M2.MNU** supplied with TopSpeed Modula-2, you can copy **vidm2.MNU** from your VID release disks over the working **M2.MNU** in your TopSpeed environment. *Do not* do this if you've made changes in **M2.MNU**.
- You can also make the changes directly in **M2.MNU**, using an editor.

After you've installed VID in the environment, you'll see a new command in the menu.

Compiling Modules to Debug

Before running VID on a program you need to ensure that you have correctly compiled the program modules you wish to debug. If you wish to debug a module, set the following compiler options to ON before compiling the module:

- **D(efug)** — to generate the **.DBD** file required by VID.
- **V(olatile)** — to ensure that variables are saved to memory (i.e., not kept in registers).
- **N(umbers)** — to generate line numbers for the module in the map.

These can be set to ON through the **Options Compiler** menu within the environment. To set the options using the batch command line use the following command:

```
m2/c filename /dvn
```

If you don't need to debug all your modules, set the preceding options to OFF for those modules. This will ensure that the modules will not "get in the way" when you begin to debug. Minimizing the number of modules being debugged also means that the amount of information given to VID is decreased. VID will run faster as a result.

Linking Your Program

When all required modules are compiled, you need to link the program and generate a map file. To do this, set the following two options to ON for the linker:

- M(ap) — generate a **.MAP** file (read by VID)
- C(ase sensitive) - make a case sensitive map

If you're using batch commands, you can use the following command:

```
m2/l filename /m/c
```

Running VID

After you have compiled and linked your program, the following files should exist and should be accessible to VID:

- progname.EXE
- progname.MAP
- modulename.MOD — for each module that you want to debug
- modulename.DBD — for each module that you want to debug

VID can locate these files through the redirection file, **M2.RED** (see the TopSpeed Modula-2 *User's Manual*).

You are now ready to invoke VID. From the DOS command line, the format of the command is as follows:

```
VID [/option] progname [progcommandline]
```

For example, the following are all valid commands lines to invoke VID:

```
vid viddemo
```

```
vid /s3 grtest
```

```
vid/s0/f255 myprog myparams
```

You'll learn what these options mean very soon.

Running VID from within the TopSpeed Modula-2 Environment

If you installed VID in the TopSpeed environment, as described earlier, you can type **AltV** to start the debugger. You'll be prompted for the main file name.

Actually, when run from the environment, VID will first leave the TopSpeed environment automatically — prompting you to save any edited files. The debugger then will proceed to run as a normal DOS program.

There are two major reasons for having VID leave the environment:

- **Safety:** So a runaway program cannot corrupt the environment, thereby leading to possible loss of data
- **Storage:** So that the program has the maximum possible space in which to run

On exit from VID, you'll be asked whether you wish to return to the environment. If you reply yes, then you'll reenter the TopSpeed environment, with the file currently on display in the VID Source window loaded into the Editor 1 window at the position of the user cursor.

After it has started running as a program, VID reads the **MAP**, **DBD** and **MOD** files. From these, VID generates a temporary symbol file that contains (in compressed form) all the information from the **MAP** and **DBD** files. This file (with the name **progrname.DB\$**) is generated only when your map file changes — that is, when your program has been remade. This saves time when starting up VID.

After VID has made the symbol file, it presents you with the normal VID view of the source of your program about to be executed. Also displayed is a small "Welcome" banner that displays the program name and version number. This banner will disappear after the first key is pressed.

The program is now ready for you to debug — using all of VID's facilities.

Expressions in VID

Before we start looking at the VID commands and options, it will be useful to discuss some points relating to expressions and how VID handles them.

Most of VID's commands work with variables or values. In many situations it will be convenient for you to enter expressions that evaluate to a particular value. For example, you may want to take actions depending on the value of an expression that uses several variables from a procedure you're debugging. If the expression is negative, let's say, you may want to do one thing, and if the expression is positive, you'll want to do something different.

VID allows you to:

- display the result of evaluating Modula-2 expressions
- watch expressions during execution
- to set breakpoints triggered by expressions becoming TRUE
- single step through your program until an expression becomes TRUE

VID will accept Modula-2 expressions involving most simple numeric and logical types and operators. Operator precedence is as defined in TopSpeed Modula-2, and you may use parentheses to change ordering.

Types allowed in VID expressions include:

CARDINAL/SHORTCARD/LONGCARD

INTEGER/SHORTINT/LONGINT

CHAR

BOOLEAN

Enumerated types

CHAR, **BOOLEAN** and Enumerated types evaluate the ordinal value of the variable's contents. For example, a **CHAR** variable containing 'a' has the value 97; a **BOOLEAN** variable that is **FALSE** has the value 0.

All types are converted to **LONGINT** to allow mixed types in expressions. For example, given

```
i=-2 (SHORTINT), k=5 (LONGCARD), c='0' (CHAR)
```

The following expression evaluates to 51

```
i+k+c
```

Operators allowed in expressions include

+ -

*** DIV / MOD %** (/ is short for **DIV**, % is short for **MOD**)

>> <<

< <= = >= > <> # (# and <> are synonymous)

AND & OR NOT ~ (**AND** and **&** are synonymous, as are **NOT** and **~**)

Logical results return **TRUE** (1) or **FALSE** (0) and may be combined using the logical operators **AND** and **OR** to provide complicated conditions. When supplied to expression breakpoints, such conditions allow you to set complex conditions for breaking the execution of your program. (See Breakpoint Expression).

Variables may be any of the above types which include array elements, record fields and dereferenced pointers. For example, the following is a valid VID expression returning **TRUE** or **FALSE**:

```
(n<>0) AND ((a[j]^tag<=0) OR (i*2>j DIV n))
```


A full EBNF description of VID expression syntax is as follows:

- **expr** ::= *simpexpr* [*relop simpexpr*]
- **simpexpr** ::= [*signop*] *term* { *addop term* }
- **term** ::= *factor* { *mulop factor* }
- **factor** ::= 'NOT' *factor*
 | '(' *factor* ')'
 | *number*
 | *variable*
- **relop** ::= '<' | '<=' | '=' | '>=' | '>' | '<>' | '#'
- **signop** ::= '+' | '-'
- **addop** ::= '+' | '-' | 'OR'
- **mulop** ::= '*' | 'DIV' | '/' | 'MOD' | '%' | '>>' | '<<' | 'AND' | '&'

When using expressions in VID as the object of a Watch or a GoTo command, keep in mind that the variables in an expression are “bound” to the variables in scope when you entered the expression.

If any of the variables in the expression go out of scope then the expression is deemed to have no value and is not displayed in the Watch window. In such a case, the expression will not terminate the Goto Expression command. When the variables return into scope, however, the expression’s value will become valid again and will be redisplayed in the watch window.

VID Command Line Options

You have several options that you can specify on the command line when starting VID:

/S Screen swap options: These tell VID what kind of a screen swap you need — that is, what kind of screen your program is using. The default setting is /S1, but this can be changed by using the default command line option when you install VID.

/S0 No swap  will simply refresh the screen)

- /S1** Page (hardware) swap; default for CGA text and compatibles (EGA/VGA)
- /S2** Screen (software) swap; default for monochrome
- /S3** CGA graphics swap; swap CGA graphic screen (this is slow)
- /S4** General graphics swap (EGA/VGA); (this is slow)
- /S5** Hercules graphics swap
- /S6** Two monitors/adaptors

Note that **/S3**, **/S4** and **/S5** require more memory, which means there is less space available for VID to use.

/BW This option forces monochrome mode for the video. VID will automatically detect monochrome modes. You can use this mode to force monochrome operation if you use a monochrome monitor with CGA.

/Fnnn This is used to specify the value with which to (byte) fill the program space. The default value is 0. Hexadecimal format is allowed: for example, **/F255 = /FFFFH**. If no fill is required, then **/F-** turns off fill.

/Ennn This command causes VID to run partially in expanded memory. VID will use expanded memory to buffer symbol information. This will add about 64K to the memory space available for your program to execute in. Also, debugging large programs will be faster as symbols will not need to be swapped in from disk. The number **nnn** specifies the number of 16K EMS pages to allocate VID. The minimum effective value is 4 (64K) and any value up to the capacity of the memory board may be specified (128 for 2MB). For example,

/e64

says to use 1MB of EMS.

Note: To use this option you must have an expanded memory board and a LIM EMS driver installed. If your configuration does not meet these criteria, the option will have no effect. If enough memory is not available then the maximum available is used.

Warning: You should *NOT* use the **/E** option if the program being debugged also uses EMS during its operation.

/K This "snow check" option turns on snow suppression for CGA monitors that produce flickering.

/R This forces a remake of the **.DB\$** compressed map file. Normally, this file will be remake only when the map file changes.

The Source Window

The main area of VID's screen is for displaying the source of the program being debugged. This area has a header that shows the name of the file currently displayed and the line number within this file. The line number changes if you move either of the two cursors available in the window.

The first is called the *execution cursor*, and it shows the source line about to be executed. This cursor appears with blue background on a color screen or in inverse video on a monochrome monitor.

The second is called the *user cursor*, and is the cursor you use to move around physically within the source. You also use this cursor to show VID where you want to put traps, etc. The user cursor is red on a color monitor and is an underline on a monochrome monitor. When the two cursors coincide on the same line, the Execution Line takes precedence.

The source file text is displayed within the central area of the debugger's screen. You can use normal cursor keys to move around within the source text. When you move around in the text, you'll observe the two bar cursors highlighting lines of text.

This area is also where the monitor count is displayed when a "monitor" breakpoint is set.

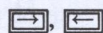
You have a rich collection of commands available for moving around in your source program and for controlling its flow. In the next few pages, we'll summarize the commands available.

Commands for Moving in the Text

You can use the following keys to move the *user* cursor around in the text:



Move the user cursor up or down a single line.



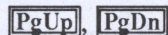
Scroll the source window to the right or to the left.



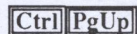
Return source window to the left edge.



Ensure that the right edge of the text is within window.



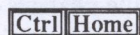
Move the user cursor up or down a page through the source.



Move the user cursor to the first line of current module displayed.



Move the user cursor to the last line of current module displayed.



Move the user cursor to the location of the execution cursor.

The movement resulting from these commands is based on the physical location of lines. This is in contrast to movements based on the flow of the executing program, as in the next set of commands.

Commands for Moving in the Running Program

The following keys have an effect on program execution and affect the *execution* cursor.

[Space]

Pressing the space bar will single step — that is, execute — a single *action* in your code, and will then move the execution cursor accordingly. After the single step the user cursor and the execution cursor will be on the same line. Note that the “size” of an action depends on whether or not the Assembler trace window is open. If this window is open, an action will consist of a single line of Assembler code; if the window is not open, an action will be a single line of Modula-2 code.

[V]

This command swaps to the output screen of the program being debugged. This allows you to see any output that the program has produced up to the current point in the program. The different screen swap options will allow swapping between the various modes of display — for example, text & graphics and so forth — as described earlier. To get back to VID’s screen when you have swapped to the program’s screen, just press any key.

If screen swapping is disabled — that is, if option */S0* was used — then **[V]** command will not swap screens but will refresh VID’s screen. This ensures that any information on the screen overwritten by the program is redisplayed.

[P]

This key moves the execution cursor, as does **[Space]**. Suppose the current line contains a procedure call. Recall that pressing **[Space]** moves the execution cursor to the first executable line of the procedure. If you press **[P]** instead of **[Space]**, the execution cursor will skip over the procedure call. The procedure will still be executed, but VID will not trace through the execution. If you skip over the procedure, any breakpoints you had set will remain active, so they won’t be missed.

[Ctrl] [Break]

This command will pause the program being debugged and will return control to VID. You can use this command when the program tracing is continuous (see Go Trace below) or when the program is executing without VID tracing. Note that VID cannot break into

DOS, so if your program is waiting for input you may have to press a key (or **Enter**) before the program is paused.

If the program stops in a module that is disabled or that is not being debugged, then the execution cursor will show the “nearest” calling point in an enabled module’s source. Together with the Procedure Trace window (described later in the chapter), this information will enable you to find the point the program was executing when **Ctrl Break** was pressed.

Commands for Handling Windows and Screens

VID is a multi-window system that can have many windows on the screen at the same time. These windows can overlap, so only the top window is guaranteed to be totally visible. The top window is called the *active window* and can be differentiated from the other windows by the fact it has a double frame while all others have single frames.

To close the active window, press **Escape**. This removes the window from the screen (uncovering anything beneath), and also terminates the function associated with the window — tracing or input, for example. The next window “beneath” then becomes the new active window.

You can reorder trace windows on the screen, by using commands described in a later section. With the appropriate command, you can bring any window to the top. You can also “cycle” most windows — that is, bring the bottom window to the top — by pressing **F6**. Input windows (for example, menus and Examine windows) cannot be cycled. They must remain the active window until they are closed by **Escape**.

Escape

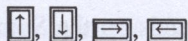
Close the last window activated.

In addition to opening and closing windows, you can move them, and can even save them to disk or send them to a printer. To move windows around on the screen — in order to uncover information hidden beneath — you need to enter the window control mode. Press the **Scroll Lock** key to do this. In this mode the cursor keys will move the active window around. When you have positioned the window in the correct place, press **Scroll Lock** again to get out of window control mode.

The window control mode also provides facilities for dumping either an entire screen or the active window to a disk file or printer. This logging capability allows you to easily append the results of debugging to a capture file for future reference.

Scroll Lock

Enter window moving mode. In this mode, you can move windows around on the screen, and you can dump a window or screen to a file or to the printer. The following commands are available in this mode.



These keys move the top window in the specified direction. In this way you can relocate the windows in your debugging environment.



This command dumps the contents of the active *window* to the current dump file or to the printer.



This command dumps the contents of the entire *screen* to the current dump file or to the printer.

If no dump file has been specified, you'll be prompted for such a file. If you enter the file name `PRN`, VID will send all dumped text to the printer. If the dump file already exists (from a previous session) you will be asked whether you wish to Append to this file, Overwrite the file or Quit the function.

Once you specify a dump file, subsequent window or screen dumps during the same session will go to the same target. For example, if you had specified a file named `DBGDMP` as the target, the contents for each `W` or `S` command in window moving mode would be appended to file `DBGDMP`.

If you want to send each dump to a different target, you need to specify a new target name between dumps. The following command lets you do this.



This command lets you specify a new target name. Material from subsequent calls to `W` or `S` will be appended to the new dump file. Remember, you must be in window moving mode for this command.

Note that you do not have to close the dump file, so that even if your program should crash the machine, your logged data will remain intact.

Commands for opening various windows are discussed in the appropriate sections.

Commands for Exiting VID

You can terminate a VID session and return to the DOS environment at any time. To exit VID, you can press `Q` or `AltX`.

Additional Commands for Moving in the Source File

In addition to the commands available for moving on the screen, you can move between procedures in the source file using the `F7` and `F8` keys. `F8` takes you to the next procedure called after the current cursor position; `F7` takes you to the previous procedure.

You can also specify these commands on the Source menu: type **[S]** for the Source menu, and then select **[N]** or **[P]**, respectively, to move to the next or previous procedure. In each case the cursor is taken to the first procedure line that has executable code. (You can also get to the Source menu by means of the main menu, **[F10]**.)

You can use **[F9]** (or **[L]** on the Source menu) to find a specific procedure. You'll be asked for the name of the procedure to be found. You can enter a procedure name, and VID will move the cursor to the beginning of that procedure. Or you can press **[Enter]** to get a list of the procedures in the current module on display (without qualification) or in other modules (with qualification).

You can also specify wild cards. In that case, VID will open a selection window from which you can select the required procedure. Note that **'**'** will match all procedures in the current module on display while **'*. *'** matches all procedures in all modules.

You can even find a particular string in the source text of the current module. This string may contain wild cards but must be on a single line. To do this, press **[Ctrl][Q][F]** (or select **[F]** from the Source menu).

If you need to check the contents of another file while running VID, you can use the **Source View** file command (**[V]** on the Source menu) to call up this file. The file can be any text file — it need not be a Modula-2 source file. You can use the cursor keys to move through this file, but you can only read the file. When you're finished with the file, just press **[Escape]**.

Highlighting and Marking Lines

You've seen that the line corresponding to the current execution cursor position is highlighted. Although we've been taking it for granted that this is the case, the execution cursor is highlighted only because the **Option Highlight** is set to ON.

When this option is ON, any source line for which the compiler has generated code can be highlighted. What this means is that the execution cursor can be moved to any line for which there is code. This shows where traps may be placed in the code — since you cannot place a trap where there is no code, as it will never be executed!

You may find sections of code that are not highlighted — even though you would expect these sections to be executable. The code may not be highlighted because smart linking or smart compilation made it unnecessary ever to execute those sections of code. Checking the highlighting of your source often proves illuminating while debugging, since you can see exactly what code is and isn't used.

Note that VID only deals in whole lines. If you have multiple statements on a single line then they will be treated as a single unit for the purpose of traps, etc. This was done to make the selection and display of lines much simpler for the user. If you

have multiple statements that you need to differentiate, then you will have to adjust the source to ensure that each statement is on a separate line.

To the left of the source, two columns are reserved for breakpoint indicators. These show the position and type of breakpoints set on the line to the right. The following indicators are used:

- ▶ One shot breakpoint
- ◆ Sticky breakpoint
- n▶ Delayed breakpoint with *n* to go ($n < 10$)
- *▶ Delayed breakpoint with 10 or more to go
- n◆ Delayed sticky breakpoint with *n* to go ($n < 10$)
- *◆ Delayed sticky breakpoint with 10 or more to go
- ?◆ Expression breakpoint
- U◆ User breakpoint
- Monitor breakpoint

See the breakpoints section below for more details on the different types of breakpoints and how they are set.

VID's Menu Structure

VID has a two-level menu system. That is, the debugger has a main menu level, whose selections you can see by pressing **F10**. This menu is shown in Figure 3-3.


```

Main menu
B - Breakpoint menu
E - Examine menu
G - Go menu
S - Source menu
T - Trace menu
O - Options menu
/ - swap screen
Q - Quit VID (AltX)

```

```

line: 92

```

```

Over,Player :

```

```

an *)

```

```

hOver.machine :

```

```

(* display text introduction *)

```

```

PushOver.Init : (* initialize pushover variables *)

```

```

PushOver.DrawBoard : (* and draw the empty board *)

```

```

k := IO.RdKey() ; (* wait for user to finish reading *)

```

```

Window.Close(IW) ; (* close introduction window *)

```

```

LOOP

```

```

PushOver.PlayGame(WhoPlaysFirst, exit) ;

```

```

IF exit THEN

```

```

VID  Keys: Space-Step P-Skip / Screen swap Esc-Close window Q-Quit VID
V1.00 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main

```

3.3. Main VID menu.

These selections represent the first menu level. All but the bottom two commands put you in a second menu. You can also reach a first-level menu directly — that is, without going through **F10** — by pressing the first letter of the menu's name. For example, press **B**. After a second or two, the Breakpoint menu pops up.

This menu contains the second-level selections. Again, you can select a command simply by pressing the letter associated with the command.

The delay before the pop-up menu appears makes it possible for you to speed up the process of selecting a second-level menu choice. For example, suppose you wanted to move the execution cursor to the current user cursor. Instead of pressing **G**, waiting for the Go menu to come up, and then pressing **C**, you can simply press **G** and **C** right afterwards. VID will bypass the menu presentation, and will execute the command directly.

You can give any command using one or two keystrokes. Once you're familiar enough with the commands, you won't ever need to use the menus.

As you saw earlier, a number of common VID commands have been assigned to single keys. You can see a summary of these keys and their functions while running VID. Just press **F1**. This opens a "key help" window which gives a short description of all the VID keys. After you press **F1**, your screen will look like the one in Figure 2-1.

Press any key to close the window. If you press one of the command keys, the help window will close and that command will be executed. If you just want to close the window without invoking a command, press **[Escape]**.

The next few sections provide a summary of the VID menus, and of the commands available in each of these menus. First, we'll look at VID's main menu.

VID's Main Menu

As you saw earlier, you can press **[F10]** if you want access to the main VID menu system. From here, you can reach all the submenus, although each is accessible by a more direct command. The main menu also contains the commands **[Q]** to exit VID and **[Z]** for screen swap – both of which were mentioned earlier.

To select an entry on the main menu, or on any other menu in VID, you can do either of the following:

- Move the bar cursor to the desired entry and press **[Enter]**.
- Type one of the characters that are displayed and highlighted. This will immediately invoke that selection. To leave a menu or submenu without making a selection, press **[Escape]**. This will close the menu window and return you to the window in which you were before opening the menu.

The main menu lists six submenus and two other commands. The selections available from the main menu are:

- [B]** Breakpoint menu
- [E]** Examine menu
- [G]** Go menu
- [S]** Source menu
- [T]** Trace menu
- [O]** Options menu
- [Z]** Swap screen
- [Q]** Quit VID (**[Alt X]**)

Each of the menus is described in detail in a later section — along with the commands available through that menu.

The **[Z]** command swaps between VID's screen and the program output screen.

The **[Q]** command will quit from VID and put you back at the DOS command prompt. If you are in the midst of a debugging session, you will be asked to verify

that you want to exit from VID. This is to prevent you from unwittingly leaving the environment by accidentally entering the wrong command.

You can also use **Alt X** as a short cut key to exit. This provides compatibility with the TopSpeed user environment.

Another way to leave the environment is to use **Go eXit** (**G X**). This command will take out all breakpoints, but will continue running the program being debugged until its conclusion.

If you called VID from within the TopSpeed environment, you'll have the option of returning to that environment when you exit VID. If you specify that you want to return, then you'll reenter the TopSpeed environment, with the file currently on display in the Source window loaded into the Editor 1 window at the position of the user cursor.

The Breakpoint Commands

Introduction to Breakpoints

Breakpoints are “software traps” that you can insert at any statement within your program — provided the statement has executable code associated with it. A software trap will pause your program when it is either tracing or executing continuously, and will then transfer control to VID. Depending on the type of breakpoint, VID will then either allow you to examine, single step or restart the program (using one of the “Go” commands), or VID will perform some breakpoint counting operation.

You can set any of several types of breakpoints. These types are: One-shot, Sticky, Delayed, Delayed/Sticky, Expression, User and Monitor breakpoints.

You can have only one breakpoint on each source line — even if you have multiple statements on that line. If you need to set independent traps on these statements you must separate them onto different lines within the source.

When setting a breakpoint on a line, any previous trap present on that line is automatically removed before the new breakpoint is set. This makes it very easy to change the type of breakpoint.

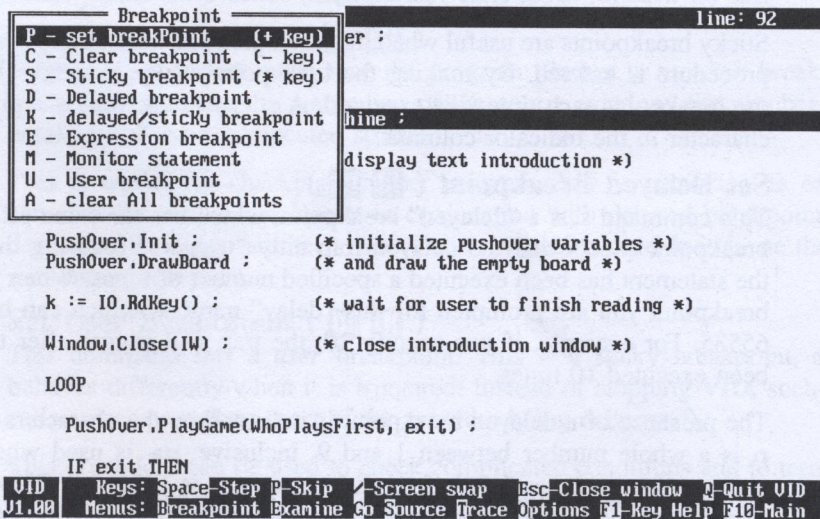
To set (or clear) a breakpoint you merely “point and shoot.” This involves first moving to the line on which you wish to place the breakpoint, and then giving the appropriate breakpoint command.

You can use cursor/page movement and/or search commands (as described earlier) to move through the source. You may also need to change modules if you want to put the breakpoint in a module other than the one on display. (To change modules, you can use **Source Change**, or **F3**, as described below.)

Once the user cursor is positioned on the target line, then the commands described below can be used to set (or in the case of **[C]**, clear) any type of breakpoint. The three most common operations — setting a one-shot breakpoint, clearing a breakpoint and setting a sticky breakpoint — have been assigned single keys namely +, - and *, respectively.

The Breakpoint Menu

The Breakpoint menu — accessible by pressing **[B]** — contains the commands shown in Figure 3-4, to enable you to set any of the types of breakpoints mentioned earlier:



3.4. The Breakpoint menu.

Set Breakpoint (**[+]** or **[B]** **[P]**)

This sets a "one-shot" breakpoint on the statement at the user cursor. This breakpoint will cause execution to stop when the statement is executed next. Once the statement executes, the breakpoint is automatically removed.

When the program "hits" a breakpoint, note that VID pauses the program *before* the statement is executed. To execute the action of the statement, you can single step through it using the **[Space]** bar.

When a statement has a “one-shot” breakpoint set on it, the ► symbol is displayed to the left of the statement text. This indicator remains attached to the statement while the breakpoint is in effect.

Clear Breakpoint (or)

This command will remove any breakpoint set on the source line at which the user cursor is currently resting. You can remove any type of breakpoint with this command. Once removed the breakpoint indicator will disappear from the left columns, indicating that there is no longer a breakpoint on the line.

Set Sticky Breakpoint (or)

This command sets a “sticky” breakpoint at the user cursor. A sticky breakpoint has the same properties as a one-shot trap except that it is not removed when the statement has executed. This means the sticky breakpoint will continue to trap the line on which it is set until you explicitly remove the breakpoint.

Sticky breakpoints are useful when, for example, you want to trap every time a certain procedure is entered. By making the breakpoint sticky, you save having to reenter the breakpoint each time it is executed. A sticky breakpoint is represented by the ♦ character in the indicator columns.

Set Delayed Breakpoint ()

This command sets a “delayed” breakpoint, which has the same effect as a one-shot breakpoint except that the delayed trap only “triggers” (pausing the program) after the statement has been executed a specified number of times. When you set a delayed breakpoint you are prompted for this “delay” number, which can be anything up to 65535. For example, if you specify 10, the trap will trigger after the statement has been executed 10 times.

The presence of a delayed breakpoint is indicated by the characters *► or n► where n is a whole number between 1 and 9, inclusive. n► is used when there are nine or fewer occurrences left before the breakpoint is triggered. This counter will be continuously updated if you are single stepping or tracing the program.

When the breakpoint is finally triggered, after the *n*th execution, the breakpoint will be removed automatically. Such a delayed trap is very useful for debugging loops, to ensure that you get the right iteration without having to step through all the execution preceding the iteration in which you’re interested.

Set Delayed/Sticky Breakpoint ()

This type of breakpoint is like a delayed breakpoint except that the sticky version is not removed after it has been triggered. Rather, the sticky breakpoint is reset to its original countdown value. This can save you the trouble of having to reenter the original breakpoint when carrying out the same function repeatedly.

As with an ordinary delayed breakpoint, you need to specify a delay value. Such breakpoints are indicated by *♦ or by n♦, depending on whether the delay value (*n*) is greater than 9.

Set Expression Breakpoint (**[B] **[E]**)**

This command prompts for an expression and then sets a sticky conditional breakpoint at the user cursor position. This is like an ordinary sticky breakpoint except it will only be triggered if the expression you entered is TRUE (non-zero) when the statement is executed. When the breakpoint is triggered the window containing the expression is displayed.

Since the breakpoint is a sticky breakpoint you must use the **B**reakpoint **C**lear command to remove the breakpoint. An expression breakpoint is represented by the ?♦ characters in the indicator column.

This type of breakpoint is extremely useful for checking program assertions and also for stopping the program when a certain state is reached. Because the expression is only evaluated when the breakpoint is executed, the expression breakpoint should be used in preference to **G**o **E**xpression whenever possible.

Monitor Breakpoint (**[B] **[M]**)**

This breakpoint does not pause the program. Rather, a *monitor* breakpoint reports continuously on the top status line of the source window, the number of times the breakpoint has been executed so far.

This is useful for checking whether program flow is occurring as expected. You can also use such a breakpoint in conjunction with delayed breakpoints in order to determine the delay number to set. The ● symbol is used to indicate the presence of a monitor breakpoint.

Set User Breakpoint (**[B] **[U]**)**

This command sets a *user* breakpoint. This is a sticky breakpoint, except that it behaves differently when it is triggered. Instead of stopping VID, such a breakpoint calls a user procedure (part of the program being debugged).

This procedure can be used to check complicated conditions and to terminate VID if these conditions are met. Also VID output routines can be called to display data in the User window. (See the advanced debugging section in Chapter 4 for more details on using these features.) If no user procedure has been set up, then the user breakpoint will have no effect. The symbols to indicate the position of a user breakpoint are U♦.

Clear All Breakpoints (**[B] **[A]**)**

This command provides a quick way of clearing all breakpoints currently set in your program. After such an action, the program is free to continue without hindrance. Such a breakpoint can be used in conjunction with the **G**o **R**e**S**tart command to restart your program with a “clean slate”.

The Examine Commands

The Examine commands allow you to look at your program's data. You can even change these data, regardless of whether they are in variables, memory locations or processor registers.

The data are presented in a "spreadsheet-like" window, which is opened when you give an examine command. Within this window there are a number of rows and columns with a single cell for each data element.

One of the cells is shown in inverse (possibly with a flashing cursor in it). This is called the *cursor cell*, and refers to the currently selected data element. If the cursor is flashing within this cursor cell, then you may edit the value displayed and thus change its value. If you accept the edit (by pressing **Enter** or moving to another cell), the new value will be written back into memory.

To move between cells (and pages of cells) use the usual cursor and page movement keys.

Values are displayed in symbolic form, and you can also enter them in this form. For example, put the the execution cursor near the top of the **VidDemo** module, then type **E G**. The Examine window will contain information about the **VidDemo** global variables. Notice that variable **WhoPlaysFirst** has the value **machine**, which is a predefined symbolic value.

Press **Escape** to close the window. In **VidDemo**, move the execution cursor to the statement that calls procedure **PushOver.DrawBoard**. Press **Space** to trace into this procedure. Now type **E G** again, to see the variables available in this procedure. Such a window is shown in the following figure.


```

PushOver Globals line: 366
b      ARRAY
BoardWin NIL      d :
PC      1          R:
LM      24
SE2D:0000 ;
IF BoardWin=NIL THEN
  BoardWin := Window.Open(
    Window.WinDef( 40,2,
                  56,16,
                  Window.Black,Window.LightGray,
                  FALSE,FALSE,FALSE,
                  TRUE,Window.DoubleFrame,Window.LightGray,Window.Blue)) :
  Window.SetTitle(BoardWin,'PushOver',Window.CenterLowerTitle) ;
ELSE
  Window.Clear ;
  Window.TextColor(Window.Black) ;
END ;
IO.WrLn;
IO.WrStr(' ');IO.WrLn;
IO.WrStr(' ');IO.WrLn;
IO.WrStr(' ');IO.WrLn;
IO.WrStr(' ');IO.WrLn;
IO.WrStr(' ');IO.WrLn;

```

Examine/modify variable  follow Esc-exit

3.5. The Examine window.

Notice that variable **b** is an **ARRAY**. (It actually turns out to be a two-dimensional array.)

Structured types (such as **ARRAYS**, **RECORDS** and **SETS**) and pointers may have subelements that are not immediately accessible and that are not displayed. You can “follow” such variable types to get to the subelements.

To do this, first move to the variable cell and press **Enter**. The display changes to show the new selections’ contents. The value of the desired subelement may be displayed. If not, it may be necessary to “follow” again.

For example, suppose you wanted to look at cell **b[2][0]**. You would select **b** as the variable of interest, by moving your cursor to the appropriate line, and pressing **Enter**.

On the resulting screen, you would select array 2 as the first required subelement, and would press **Enter** again. After this, your screen will look like the one in Figure 3-6:


```

b[2][0]
[0]    opponent
[1]    opponent
[2]    opponent
[3]    opponent
[4]    opponent
5E2D:0012
Line: 366

BoardWin := Window.Open(
    Window.WinDef( 40,2,
        56,16,
        Window.Black,Window.LightGray,
        FALSE,FALSE,FALSE,
        TRUE,Window.DoubleFrame,Window.LightGray,Window.Blue)) ;
Window.SetTitle(BoardWin,'PushOver',Window.CenterLowerTitle) ;
ELSE
    Window.Clear ;
    Window.TextColor(Window.Black) ;
END ;
IO.WrLn;
IO.WrStr(' ');IO.WrLn;
IO.WrStr(' ');IO.WrLn;
IO.WrStr(' ');IO.WrLn;
IO.WrStr(' ');IO.WrLn;
Examine/modify variable  ←-accept Esc-exit

```

3.6. Examine window showing cells of a two-dimensional array.

You can use the cursor keys to move to the individual cells of `b[2]`. The indicator at the top of the window tells you what cell you're currently examining.

To close a variable display, press **Escape**. This takes you back to the "parent" display from which you selected the variable. If there is no parent display then **Escape** will close the Examine window. Note that following **POINTERS** will not back up to the pointer in the "parent" display when you press **Escape**. This allows you to follow an arbitrary number of pointers without having to retrace your steps back to the first.

Variant records have all variants displayed independent of the tag field. This means that changing one field may have an effect on other fields in the variant record.

For all memory display types the address of the current cell in memory is displayed in the lower right corner of the window's frame. The title at the top of the display window shows what object or objects are currently being displayed and in the case of array elements shows the current cell's array index. The leftmost column of the display also gives information on the contents of each row in the display.

ASCII strings (that is, **ARRAYs OF CHAR**) are displayed (and input) as a contiguous line of characters. To change the characters you can just type over the material contained in the cell.

If you need to enter a non-printing character, press **AltN**. A window will appear, prompting you for the ASCII code of the character you wish to write at the current

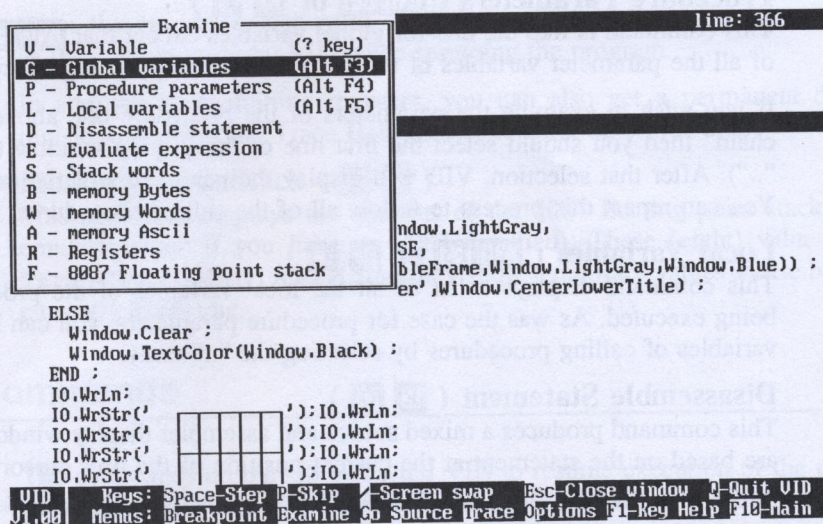
cursor position in the string. The character you specify will replace the character in the variable.

To enter the NULL character (ASCII `CHAR(0)`), type `[Alt][0]`. When present in the string, this character is displayed as `■`.

For cardinal and integer (normal, short and long) variables, you can enter expressions to calculate new values for the variables. See the section on expressions earlier in the chapter for more details on the format valid expressions must have.

Examine Menu

The commands accessible through the Examine menu are shown in the next figure.



3.7. The Examine menu.

Examine Variable (`[?]` or `[E][V]`)

This command prompts for the name of a variable to display. You may specify any variable currently in scope or a variable prefixed with the module name (for example `mod.variable`). By prefixing a variable name with a module name, you can access global variables belonging to modules other than the one currently on display. For example, `FIO.EOF` will access the variable `EOF` in the module `FIO`.

When you specify the name and press `[Enter]`, the variable value(s) will be displayed, ready to be changed if necessary. The display window title indicates the variable's

“owner.” This will be either a procedure (for local variables) or a module (for global variables).

You can also enter valid Modula-2 expressions instead of a variable. In that case, the evaluated expression will be displayed. See the section on expressions for further details.

Global Variables (`Alt F3` or `E G`)

This displays a list of all the global variables of the module currently on display in the source window. Structured variables may be selected as described above — by opening a full display window. Other variables will have their values displayed in the list. These values can be changed by moving the bar cursor to them and editing the value.

Note that you can only access the global variables of modules that are enabled for debugging.

Procedure Parameters (`Alt F4` or `E P`)

This command is like the one for global variables except that `Alt F4` displays a list of all the parameter variables of the procedure currently being executed.

If you wish to examine the parameters of the procedure one above in the “calling chain” then you should select the first line of the display window (the line marked “..”). After that selection, VID will display the parameters of the calling procedure. You can repeat this process to follow all of the calling procedures’ parameters.

Local Variables (`Alt F5` or `E L`)

This command displays a list of all the local variables of the procedure currently being executed. As was the case for procedure parameters, you can look at the local variables of calling procedures by selecting the “..” entry.

Disassemble Statement (`E D`)

This command produces a mixed source and assembler display window. The contents are based on the statement at the current position of the user cursor. You may page backwards and forwards through the display, throughout the current module.

For a more comprehensive disassembly to a text file, the utility **M2DASM** has been included with your debugger. See Chapter 4 for further details.

Evaluate Expression (`E E`)

This command displays the value of an expression entered. See the description of expressions for more information about what you can enter, and the format this must have.

Stack Words (`E S`)

This displays the stack as hex words based from **SS:SP**.

Memory Bytes (☐ ☐)

This prompts for an address (or the name of a variable on which to base the address), and then displays the contents of memory as hex *bytes*.

Memory Words (☐ ☐)

This prompts for an address (or the name of a variable on which to base the address), and then displays the contents of memory as hex *words*.

Memory ASCII (☐ ☐)

This prompts for an address (or the name of a variable on which to base the address), and then displays the contents of memory as *ASCII characters*.

Registers (☐ ☐)

This allows you to display and change the processor registers. Registers are displayed as hex words. The following registers are displayed:

AX, BX, CX, DX, SI, DI, DS, ES, BP, SP, SS, CS, IP and FL

FL is the processor flags word. Note that CS, IP, SS and SP may not be changed as they are restored by VID when resuming the program.

In addition to examining registers, you can also get a permanent display of the registers by using the **Trace Registers** command, described later.

Floating Point Stack (☐ ☐)

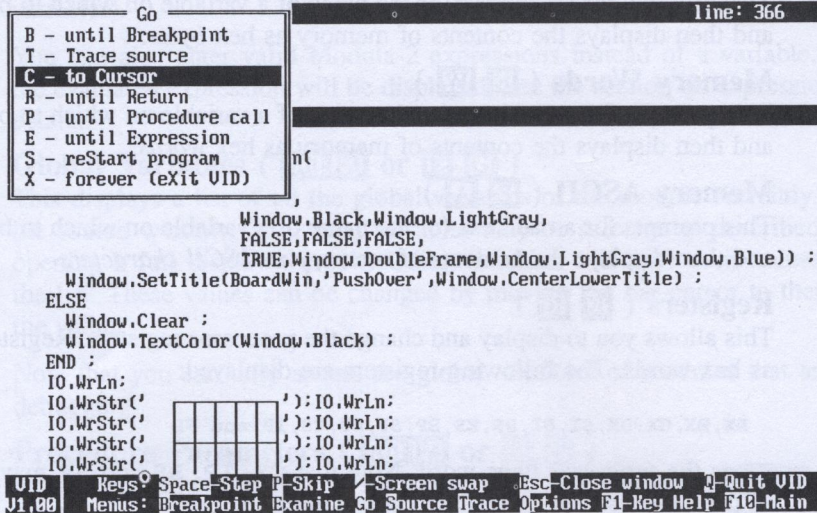
This command displays the contents of the 8087 floating point stack (or the 8087 emulator stack if you have no 8087 installed). These (eight) values are in 8087 **TEMPREAL** (10 byte) format, and they show temporary values in use by the program currently executing.

The Go Commands

The commands on the Go menu tell VID to resume execution of the program being debugged. Each command either specifies a termination condition or the mode of execution (in the case of **Go Trace**).

The Go Menu

The following figure shows the menu when you specify ☐.



3.8. The Go menu.

Go until Breakpoint (**G** **B**)

This command causes execution of the debugged program to continue until either an active breakpoint is hit or until you press **Ctrl+Break**. When this happens the execution cursor in the source window shows the point you have reached in your program — that is, the line containing the next statement to be executed.

The program executes at full speed until it reaches the breakpoint, as execution is neither being stepped nor traced.

Go Trace Source (**G** **T**)

This is the same as the **Go until Breakpoint** command except that VID displays each source line (by moving the execution cursor in the source window) as it executes on the way to the breakpoint.

The result is that the program's flow is visibly traced through the source executed. Because the display is updated quite rapidly, tracing at full speed may be too fast to see what is happening, so an option is provided to slow down the stepping speed. See **Options Debug** for details.

Go to User Cursor (**G** **C**)

This command is the same as **Go until Breakpoint** except that this command also installs a (temporary) breakpoint at the current user cursor position. This means that the program will stop when it gets to where you are "pointing" with the user cursor — provided the program doesn't stop elsewhere first, for any other reason.

This command provides a quick way of moving around your program: Use cursor movement commands to move to the statement you want, then select **Go Cursor** (or type **G C**) to move the execution cursor.

Go until Return from Current Procedure (**G R**)

This is also like **Go until Breakpoint** except that it places a temporary trap on the statement *following* the return from the current procedure — that is, on the statement after the place where the current procedure was called.

This means that, when the procedure finishes (or earlier if another breakpoint was hit), control will be returned to VID. This is very useful if you have accidentally “stepped into” a procedure through which you didn’t want to step. For example, you might have pressed **Space** instead of using **P** to skip the call. Type **G R** to return to the main thread that you were debugging.

Pressing **Go Return** while in initialization code will skip to the next initialization body. If the module is the main module, this command will make the program go until **halt**, and you will be prompted to make sure you want to do this.

Go until Procedure Call (**G P**)

This command moves the evaluation cursor to the start of the procedure you specify. You’ll be asked for a procedure name. If you press **Enter**, you’ll see a list of accessible procedures. You can use the cursor keys to move to your selection. Press **Enter** to confirm your selection.

You can also specify a name — including any necessary qualifiers. If you specify a module name and a wildcard — for example, **MyModule.*** — VID will display a list of accessible procedures from the module you specified.

Go until Expression True (**G E**)

This command first prompts for an expression, as described earlier in this chapter. After you’ve entered an expression, this command will step the program until that expression is **TRUE** or non-zero (or until either a breakpoint or **Ctrl Break** occurs).

Control then returns to VID and you are prompted with the expression and its result (> 0 or **TRUE**). Press **Escape** to continue debugging.

Although VID expressions are partially compiled and are evaluated quite rapidly, the **G E** command will still require a lot of recalculations. This may slow VID down considerably. It is always better to use expression breakpoints wherever possible.

Restart the Program (**G S**)

The **Go ReStart** command will restart the program, by reloading the program and resetting execution to the start of the program. All breakpoints are retained.

This command is very useful when you inadvertently execute a little too far in your program and you want to try again. It is also useful after you’ve learned more about what’s going on in the program.

For example, after the Monitor breakpoint has shown how many times a statement is executed, you can restart the program and set a delayed breakpoint to go just “before” that number of times. To clear all breakpoints after restarting the program you can use the **B**reakpoint clear **A**ll command.

Go Forever (**G** **X**)

This command removes all breakpoints and then resumes execution of the program. It is generally used when you have finished debugging and want to run the program to completion.

The Trace Windows

Trace windows are the windows through which you can view various aspects of your program as it executes. These windows are persistent in that they remain on the screen while you execute through your program. This is in contrast to windows such as the Examine window, which must be closed before you can continue.

The largest trace window is the Source window, which always remains open in the background. All the other trace windows can be opened and closed at any time. You can also move these windows around and can cycle through them, as described earlier.

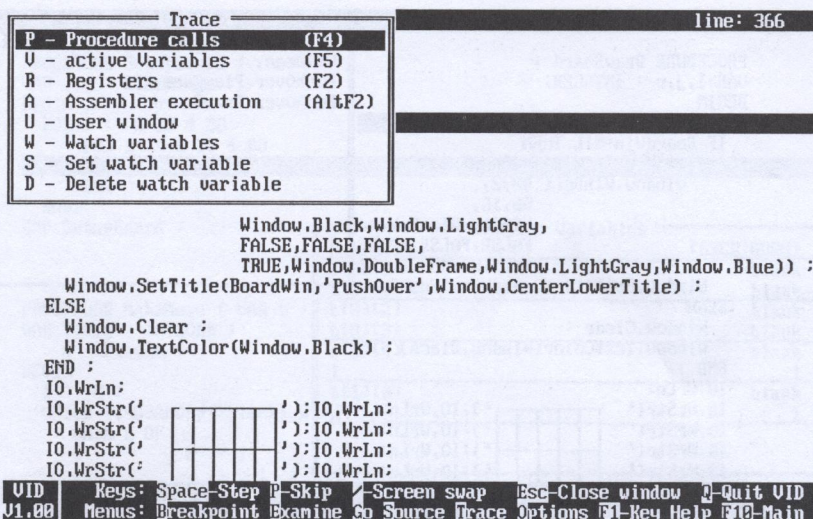
To close the top window, press **Escape**. This stops the trace function active in the window, and removes the window from the screen. The following trace windows are available, in addition to the Source window:

- Procedure call stack trace.
- Active variables trace.
- Registers trace.
- Assembler execution trace.
- User trace.
- Watch variable trace.

Any combination of trace windows may be opened; these windows will all function simultaneously.

The Trace Menu

The commands available through the Trace menu are shown in the following figure. As with the other menus, you can have a menu pop up by pressing **T** and waiting a second or two. Or you can activate the following commands by pressing **T**, followed immediately by the appropriate next character.



3.9. The Trace menu.

Procedure Calls (**F4** or **T P**)

This command displays the current “call stack” — that is, the chain of procedure calls that led to the current position. Each line contains the name of the calling procedure and (except for the current procedure) the line number of the module source where that call to the next procedure was made.

The order of procedures is from “highest” to “lowest” — that is, the current procedure is last in the calling stack. If there is a deeper nesting of calls than the window allows (that is, more than 11 procedures), then only the final 11 calls will be displayed.

The following figure shows such a trace window for VidDemo.

File: PushOver line: 166

```

PROCEDURE SetupBoard ;
VAR i,j : SideRange ;
BEGIN
  FOR i := 0 TO 4 DO
    FOR j := 0 TO 4 DO
      b[i,j] := blank;
    END;
  END;
END SetupBoard ;

PROCEDURE MakeMove ( VAR b :
VAR i,x,y : INTEGER ;
q : Player ;
BEGIN
  i := 0 ;
  WHILE (i<5)AND(p<>blank) D
    CASE n OF
      0 : x := i ;
      1..5 : x := n-1 ;
    END;
  END;

```

Active Variables	
BoardWin	[63CB:0000]
b[0][0]	blank
b[0][1]	blank
b[0][2]	blank
b[0][3]	blank
b[0][4]	blank
i	1
b[1][0]	blank
j	1

UID 01.00 Keys: Space-Step P-Skip Screen swap Esc-Close window Q-Quit UID
 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main

3.11. Trace window showing active variables.

You will find the variable trace window extremely useful as you debug. This window can be opened quickly, to see the last few variables active without having to type their names. The variable trace window has a small “look-back” buffer so that, even when it is closed, opening the window will display the last few variables accessed. The following variable types are traced in this window:

- CARDINAL/SHORTCARD/LONGCARD
- INTEGER/SHORTINT/LONGINT
- LONGREAL/REAL
- CHAR/BOOLEAN
- Subrange types
- Enumerated types

Furthermore, elements of arrays (up to 8 dimensions), fields of records (up to 8 levels), and dereferenced pointers (up to 2 levels of indirection) will also be traced. In addition to tracing the dereferenced value of a pointer, the absolute address contained in the pointer is also displayed. If the current value is **NIL**, then this will be displayed.

Registers Trace (**T** **R**)

This command displays the values currently contained in the processor registers, as shown in Figure 3-12. The values are displayed as hex words. Registers traced are **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, **BP**, **DS**, **ES**, **SS:SP**, **CS:IP** and the processor flags. The flags are displayed in symbolic form if they are set, according to the following:

```

File: PushOver                               line: 166

PROCEDURE SetupBoard ;
VAR i,j : SideRange ;
BEGIN
  FOR i := 0 TO 4 DO
    FOR j := 0 TO 4 DO
      b[i,j] := blank;
    END;
  END;
END SetupBoard ;

PROCEDURE MakeMove ( VAR b : Board ; n : IN
VAR i,x,y : INTEGER ;
q : Player ;
BEGIN
  i := 0 ;
  WHILE (i<5)AND(p<>blank) DO
    CASE n OF
      0 :      x := i      ; y := i      !
      1,5 :    x := n-1    ; y := i      !
    END
  END

```

Registers

AX=0005 BX=0005 CX=0002 DX=0000
SI=0229 DI=013A BP=3FE4
DS=5E2D ES=5E30 SS:SP=5E4D:3FDC
Flags= ITS A C CS:IP=5B6F:0186

```

UID  Keys: Space-Step P-Skip Screen swap Esc-Close window Q-Quit UID
V1.00 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main

```

3.12. Registers trace window.

- C Carry flag
- P Parity flag
- A Auxiliary Carry flag
- Z Zero flag
- S Sign flag
- T Trap flag
- I Interrupts enable flag
- D Direction flag
- O Overflow flag

To *change* the values of registers, you should use the **Examine Registers** command (**E** **R**).

Assembler Trace ()

VID is designed primarily as a high-level debugger. Nevertheless, it is sometimes quite useful to be able to view the assembler code (generated by the compiler) as it executes. Following this code gives you a much better feel for what is actually happening when each statement executes.

The assembler trace function gives a scrolling display of each processor instruction as it executes. Procedure names are displayed symbolically and the values of registers used in the instruction are displayed in parentheses after the register. The address of the instruction is displayed on the left. After the assembler opcode and operands you'll sometimes find comments displayed. These identify run-time error checks and 8087 emulated code.

Figure 3-13 shows an Assembler trace while procedure **SetupBoard** is executing.

File: PushOver		Line: 166																										
<pre> PROCEDURE SetupBoard ; VAR i,j : SideRange ; BEGIN FOR i := 0 TO 4 DO FOR j := 0 TO 4 DO b[i,j] := blank; END; END; END SetupBoard ; </pre>																												
<pre> PROCEDURE MakeMove (VAR b : VAR i,x,y : INTEGER ; q : Player ; BEGIN i := 0 ; WHILE (i<5)AND(p<>blank) DO CASE n OF 0 : x := i ; y 1..5 : x := n-1 ; y </pre>																												
		<table border="1"> <thead> <tr> <th colspan="2">Assembler Trace</th> </tr> </thead> <tbody> <tr><td>5B6F:0189</td><td>MUL Word Ptr[BP-4](1)</td></tr> <tr><td>5B6F:018C</td><td>MOV BX,Word Ptr[BP-6](1)</td></tr> <tr><td>5B6F:018F</td><td>ADD BX(1),AX(5)</td></tr> <tr><td>5B6F:0191</td><td>MOV DS,CS:Word Ptr[0C15](5E2D)</td></tr> <tr><td>5B6F:0196</td><td>MOV Byte Ptr[BX+0](0),2</td></tr> <tr><td>5B6F:019B</td><td>INC Word Ptr[BP-6](1)</td></tr> <tr><td>5B6F:019E</td><td>CMP Word Ptr[BP-6](2),+4</td></tr> <tr><td>5B6F:01A2</td><td>JBE 0186</td></tr> <tr><td>5B6F:0186</td><td>MOV AX,5</td></tr> <tr><td>5B6F:0189</td><td>MUL Word Ptr[BP-4](1)</td></tr> <tr><td>5B6F:018C</td><td>MOV BX,Word Ptr[BP-6](1)</td></tr> <tr><td>5B6F:018F</td><td>ADD BX(2),AX(5)</td></tr> </tbody> </table>	Assembler Trace		5B6F:0189	MUL Word Ptr[BP-4](1)	5B6F:018C	MOV BX,Word Ptr[BP-6](1)	5B6F:018F	ADD BX(1),AX(5)	5B6F:0191	MOV DS,CS:Word Ptr[0C15](5E2D)	5B6F:0196	MOV Byte Ptr[BX+0](0),2	5B6F:019B	INC Word Ptr[BP-6](1)	5B6F:019E	CMP Word Ptr[BP-6](2),+4	5B6F:01A2	JBE 0186	5B6F:0186	MOV AX,5	5B6F:0189	MUL Word Ptr[BP-4](1)	5B6F:018C	MOV BX,Word Ptr[BP-6](1)	5B6F:018F	ADD BX(2),AX(5)
Assembler Trace																												
5B6F:0189	MUL Word Ptr[BP-4](1)																											
5B6F:018C	MOV BX,Word Ptr[BP-6](1)																											
5B6F:018F	ADD BX(1),AX(5)																											
5B6F:0191	MOV DS,CS:Word Ptr[0C15](5E2D)																											
5B6F:0196	MOV Byte Ptr[BX+0](0),2																											
5B6F:019B	INC Word Ptr[BP-6](1)																											
5B6F:019E	CMP Word Ptr[BP-6](2),+4																											
5B6F:01A2	JBE 0186																											
5B6F:0186	MOV AX,5																											
5B6F:0189	MUL Word Ptr[BP-4](1)																											
5B6F:018C	MOV BX,Word Ptr[BP-6](1)																											
5B6F:018F	ADD BX(2),AX(5)																											
<table border="1"> <tr> <td>VID</td> <td>Keys: Space-Step</td> <td>P-Skip</td> <td>Screen swap</td> <td>Esc-Close window</td> <td>Q-Quit VID</td> </tr> <tr> <td>V1.00</td> <td>Menus: Breakpoint</td> <td>Examine</td> <td>Go Source Trace</td> <td>Options F11-Key Help</td> <td>F10-Main</td> </tr> </table>			VID	Keys: Space-Step	P-Skip	Screen swap	Esc-Close window	Q-Quit VID	V1.00	Menus: Breakpoint	Examine	Go Source Trace	Options F11-Key Help	F10-Main														
VID	Keys: Space-Step	P-Skip	Screen swap	Esc-Close window	Q-Quit VID																							
V1.00	Menus: Breakpoint	Examine	Go Source Trace	Options F11-Key Help	F10-Main																							

3.13. Assembler trace window.

Watch Variables

You can trace the values of variables and expressions in the Watch window. This window can hold up to 9 separate expressions. To set a watch variable, use the **Trace Set Watch** command ( ), as described below.

To delete a watch variable, use the **Trace Delete watch** command (**T D**). The **Trace Watch** command (**T W**) will open (that is, display) the Watch Variables window. All these commands are described below.

You can watch all “simple” variable types:

CARDINAL, SHORTCARD, LONGCARD,
 INTEGER, SHORTINT, LONGINT,
 CHAR, BOOLEAN, Enumerated types,
 REAL, LONGREAL,
 POINTER/ADDRESS (address of pointer may be traced),
 ARRAY OF CHAR (first 20 chars may be watched).

This means, of course, that you can watch record fields, array elements and dereferenced pointers — provided they have one of the above types.

It is very important to note that the variables used in a watch variable or a watch expression *must* be in scope when the watch is set. These variables are then “bound” to the watch variable/expression. This means that, even if a new variable with the same name as one of the bound variables comes into scope, the original variable’s value will still be the one displayed.

When any variable component of a watch expression is not “active,” then the watch value will be blanked out as it has an indeterminate value. This can happen, for example, with a local variable belonging to a procedure not currently on the call stack.

A small example should help clarify this.

```
PROCEDURE P1 ;
VAR
  i : CARDINAL ;

  PROCEDURE P2 ;
  VAR
    i : CARDINAL ;
  BEGIN
    i := 2 ;
    (* B *)
  END P2 ;

BEGIN
  i := 1 ;
  (* A *)
  P2 ;
END P1 ;

VAR
  i : CARDINAL ;
BEGIN
  P1 ;
  i := 3 ;
  (* C *)
END Main.
```


If a watch variable is set on the variable `i` at point `(* A *)`, then the watch variable will display the value `1` at that point. Continuing execution, at point `(* B *)` the watch variable will still display `1`, even though there is a “new” `i` here, with the value `2`. This is because the `i` in the watch variable window still refers to the `i` local to `P1`.

When program execution reaches point `(* C *)`, the watch value will be blank, since procedure `P1` and its local variables are no longer active. Note that if we had set a watch variable on `i` at `(* B *)`, we could get two different values displayed for `i` simultaneously — reflecting the different variables!

Figure 3-14 shows the situation where both variables are being watched.

```
File: Main line: 13
PROCEDURE P2 ;
VAR
  i : CARDINAL ;
BEGIN
  i := 2 ;
  (* B *)
END P2 ;

BEGIN
  i := 1 ;
  (* A *)
  P2 ;
END P1 ;
VAR
  i : CARDINAL ;
BEGIN
  P1 ;
  i := 3 ;
  (* C *)
END Main.
```

Watched Variables		
1	i	1
2	i	2

```
VID  Keys: Space-Step P-Skip Screen swap Esc-Close window Q-Quit VID
V1.00 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main
```

3.14. A watch variable window with duplicated variable names.

Watch Variable Trace (**T** **W**)

This command opens the watch variable window and makes this the active window. If there are no watch variables already set, then you will be prompted to enter one.

To the left of the window, the watch number (1 – 9) is displayed. This serves to identify the watch variables when deleting them. After the watch number, you’ll find the variable/expression being watched. Finally, the value of the variable value is at the right — provided the variable is active (see above). Figure 3-15 shows such a window.

The depth of the watch window is determined by the maximum watch number currently set. This is so the watch window takes up as little space as possible on the VID display. For easier reading, the window is “striped” in a manner similar to the active variables trace window (see above).

File: VidDemo line: 49

```
WD : Window,WinDef ;
BEGIN
  WITH WD DO
    X1 := 0 ; Y1 := 0 ; (* top left corner *)
    X2 := 37 ; Y2 := 24 ; (* bottom right *)
```

```
Foreground := Window.White ; (* normal text *)
Background := Window.Red ;
```

```
CursorOn := FALSE ; (* no cursor *)
```

```
WrapOn :=
```

```
Hidden :=
```

```
FrameOn :=
```

```
FrameDef :=
```

```
FrameFore :=
```

```
FrameBack :=
```

```
END ;
```

```
IW := Window.Open(WD); (* open the window *)
```

```
UID Keys: Space-Step P-Skip Screen swap Esc-Close window Q-Quit UID
U1.00 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main
```

3.15. Watch variable trace window.

Set Watch Variable (**T** **S**)

This opens the watch window and prompts you to set a new watch variable. This variable will be added to the watch trace window in the first available free slot. The size of the window will be extended, if necessary. The maximum number of watch variables allowed is 9. When you are adding a watch variable, your screen will look like the one in Figure 3-16.

line: 49

Variable to watch:

```

WITH WD DO
    X1 := 0 ; Y1 := 0 ;           (* top left corner *)
    X2 := 37 ; Y2 := 24 ;        (* bottom right *)

    Foreground := Window.White ; (* normal text *)
    Background := Window.Red ;

    CursorOn := FALSE ;           (* no cursor *)
    WrapOn := Watched Variables
    Hidden :=


|                |           |
|----------------|-----------|
| 1 WD.X1        | 0         |
| 2 WD.Y2        | 24        |
| 3 WD.FrameOn   | TRUE      |
| 4 WD.Hidden    | FALSE     |
| 5 WD.FrameFore | LightGray |


    FrameBack :=

END ;

IW := Window.Open(WD);           (* open the window *)

```

VID Keys: Space-Skip P-Skip Screen swap Esc-Close window Q-Quit VID
 V1.00 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main

3.16. Prompt when adding a watch variable.

Delete Watch Variable (**T** **D**)

This command prompts for the number of the watch variable to delete. The watch variable with the specified number is deleted, and that slot is freed for reuse. The watch window will “shrink” if the deleted watch number was the highest in the window.

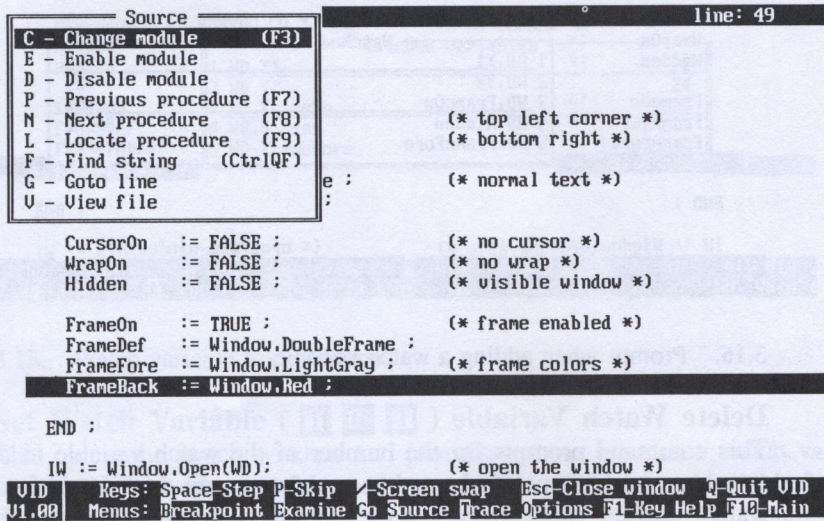
Commands for the Source Window

VID is a multi-module debugger. To aid with the debugging of multiple modules, VID provides the ability to dynamically enable and disable the modules being debugged. A disabled module will play no part in execution tracing or with respect to breakpoints. Such a module will not “get in the way” of debugging other modules.

Initially all modules that have been prepared for debugging are enabled. However, during execution you can disable or re-enable any module, by using a simple module selection window. You can also use this module selection window to change the module on display.

The Source Menu

The Source menu also contains a number of the source movement commands that you've already encountered (see *Moving in the Source*). These commands are shown in Figure 3-17.



3.17. The Source menu.

Change Module(**[F3]** or **[S][C]**)

This command displays a list of all modules from which you can select the new module to display in the source window. This is used when you want to set breakpoints in another module or you just want to look around the module.

Enable Module (**[S][E]**)

This displays a list of disabled modules from which you can select a module to re-enable. This will bring the module back into the group being debugged.

Disable Module (**[S][D]**)

This displays a list of enabled modules from which you can select a module to disable. The selected module is removed from debugging actions. The debugger will not trace through the module, and any breakpoints set in it will be inactive.

You will not be able to set new breakpoints in the disabled module until it has been re-enabled. If you are currently tracing through the module being disabled then that trace will continue. However, once you have left the module, you will not reenter.

Previous Procedure (F7 or S P)

This moves the user cursor in the source window to the start of the *previous* procedure in the module, where “previous” refers to physical proximity, not to the procedure executed previously. This command is useful for rapidly scanning through a module’s source.

Next Procedure (F8 or S N)

This command moves the user cursor in the source window to the start of the *next* procedure in the module, where “next” refers to physical proximity, not next in execution sequence. This command is useful for rapidly scanning through a module’s source.

Locate Procedure (F9 or S L)

This prompts for the name of a procedure to locate, then moves the user to the start of the procedure specified. If you simply press Enter, without specifying a name, you’ll get a window containing the accessible procedures. You can use the cursor movement keys to select the procedure to locate from this list. If you specify the name, you must qualify it with the module name, unless the procedure can be found in the current module on display.

You may use the wild cards * and ?. In that case, a selection window will open, and a list of matching procedure names will be displayed. If no module name is specified, then the current module on display is assumed. You may use wild cards for module names as well. Thus * matches all procedures in the current module, *mod.** matches all procedures in module *mod*, and *. * matches all procedures.

If the procedure you select is from another module, then that module is loaded into the source window.

Find String (Ctrl Q F or S F)

This command lets you search for a string that you specify. The command prompts for a string, and then searches for the next occurrence of that string in the currently displayed module.

The string must be within a single line. You can specify the wild cards * and ? as part of the search string. You can repeat the search as often as necessary by pressing Ctrl L.

Goto Line (S G)

This command moves the user cursor to the line number you specify. This can be helpful in conjunction with the Procedure Calls trace window. The Procedure Calls window tells you the line number at which each suspended procedure called the next procedure.

With the S G, you can quickly look at those lines, to make sure the procedures are being called correctly, etc.

View File (**S** **V**)

The view file command is provided to enable you to view files that are not among the implementation modules being debugged. For example, you may want to look at definition or documentation files for a module.

If you select this command, you'll be asked for the name of the file to display. This file is loaded, and is displayed in a full screen window. You can use the ordinary cursor keys to move around the file and examine the file's contents.

Files are not limited to 64K or to memory size. This means, for example, that you can view the contents of a large map file. While you are viewing a file, you can search for a string by typing either **F** or **Ctrl****Q****F**. This search will be done in the same manner as described above (in relation to Find String).

Pressing **Escape** will close the file and will return you to the normal VID screen. Note that you may *not* use wild cards when specifying the name of the file to view. Similarly, the DOS redirection file is *not* active. This means that, if you want to view a file in a different directory, you'll need to specify the full path to the file.

User Options

There are a number of options designed to make VID easier and more convenient to use. You can set these options at any time while running VID. You can use the **Options Save** command to keep these settings in the file **VID.CFG**. Or you can set the Auto save option, so that such settings are saved automatically at the end of the session.

The Options Menu

The settings available through the Options menu are shown in the following figure.


```

Option Menu
H - Hex output      OFF
R - screen swap     ON
V - Volatile check  ON
C - highlight Code  ON
L - status Line     ON
P - single Process  OFF
T - Trace assembler OFF
D - trace Delay     0
S - Save options
A - Auto save options OFF

WrapOn      := FALSE ;
Hidden      := FALSE ;

FrameOn     := TRUE ;
FrameDef    := Window.DoubleFrame ;
FrameFore   := Window.LightGray ;
FrameBack   := Window.Red ;

ite ;
d ;

(* top left corner *)
(* bottom right *)
(* normal text *)
(* no cursor *)
(* no wrap *)
(* visible window *)
(* frame enabled *)
(* frame colors *)

END ;

IW := Window.Open(WD);
(* open the window *)

VID Keys: Space-Skip P-Skip Screen swap Esc-Close window Q-Quit VID
V1.00 Menus: Breakpoint Examine Go Source Trace Options F1-Key Help F10-Main

```

3.18. The Options menu.

Hex Output (☐ ☒ H)

This option determines whether the default output format is decimal or hexadecimal for values displayed by VID. Some values are displayed in hexadecimal format by default (for example, register and memory values), and such variables are not affected by this option.

This option toggles between decimal and hexadecimal.

Screen Swap (☐ ☒ R)

This toggles the mode for screen swapping among two options:

- ON — this is the normal screen swap mode as specified by the /S command line option.
- OFF — this setting disables screen swapping. When this option is set, ☒ will then refresh the VID screen instead of swapping.

Volatile Check (☐ ☒ V)

Ordinarily, VID checks that you have set the /V compiler option (or (*\$W+*) in source) when compiling debug modules. When set to OFF, this option disables the check made by VID.

The volatile option ensures that variables are always kept in memory (as opposed to registers) and are thus accessible to VID. If you wish to debug without volatile set, then set this option to OFF; the checks will be disabled. However, if you turn this

checking OFF, you'll have to be careful when you examine variables as their values may be kept within registers.

Highlight Code (☐ ☐)

This enables (if ON) or disables (if OFF) the VID feature that source lines with associated code are highlighted when the execution cursor reaches the line. You may wish to disable this if the differing colors in the source window become distracting.

Status Line (☐ ☐)

This enables (if ON) or disables (if OFF) the two-line help window displayed at the base of the VID screen. If the help window is disabled then the source window is extended by two lines giving you a larger view into the source file. You will probably want to disable the help lines once you have become familiar enough with VID's keys and menus.

Single Process (☐ ☐)

When ON, this option restricts tracing and breakpoints to a single process. Thus, if you have multiple processes running, other processes will be ignored. See the section in advanced debugging for more details on debugging programs with multiple processes.

Trace Assembler (☐ ☐)

When ON, single stepping (or continuous tracing) with the assembler window open will "follow" calls into Assembler, disabled modules or into modules not being debugged. This means that you can follow the execution wherever it may lead. This option is normally set to OFF, as you are usually only interested in tracing the enabled Modula-2 modules.

Trace Delay (☐ ☐)

This prompts for the time (in hundredths of a second) to delay each single step when continuously tracing. This is useful in order to view the execution of the program at a more leisurely pace, giving yourself a chance to see exactly what is occurring.

Save Options (☐ ☐)

This command saves to disk all currently set options in the file **VID.CFG**. This also saves the positions of all the windows, along with information as to which trace windows are currently open. When VID next executes it will reload all the options and window positions from the configuration file before starting. VID can find the **VID.CFG** file using the **M2.RED** redirection file.

Auto Save Options (☐ ☐)

When set to ON, this causes options to be saved automatically whenever you leave VID — that is, the **Options Save** command is carried out automatically. This ensures that when you restart VID you will resume the session with the same options and windows configurations you had when you ended the last session.

Default Options

If you ever need to reset VID to its default options, simply delete the **VID.CFG** file. The default options are as follows:

Hex Output	OFF
Screen Swap	ON
Volatile Check	ON
Highlight Code	ON
Status Line	ON
Single Process	OFF
Trace Assembler	OFF
Trace Delay	0
Save Options	(no default)
Auto Save Options	OFF

Run-Time Errors

If your program encounters one of the Modula-2 run-time errors while debugging with VID, you are taken to the point in the source where the error occurred. (If the module in which the error occurred is not active, then you're taken to the nearest calling active module.)

You can then specify whether to continue execution of the program or to pause the program, returning control to VID. The following run-time errors are recognized:

- Arithmetic overflow
- Stack overflow
- Subrange value out of range
- Enumeration value out of range
- Dereference of **NIL** pointer
- Index out of range
- No **RETURN** from function

Note that stack overflow is fatal and execution cannot be continued.

Chapter 4

Advanced Debugging Topics

VidIf: The Program Interface to VID

For advanced tracing, VID has a program interface which allows the program being debugged to output information in a VID trace window. You can also connect a Modula-2 procedure to a “User” breakpoint that you set. This breakpoint will then call the procedure whenever it is executed. (The procedure must be part of the debugged program code.) By calling a procedure, you can test even complicated break conditions, and you can generate exactly the output you want.

The facilities for interfacing your program to VID are contained in the two source files, **VidIf.DEF** and **VidIf.MOD**.

Tracing from Your Program

VID has a tracing window reserved especially for user trace output generated by **VidIf**. This is called the *User Trace Window*. You can open it using either the VID command **[T][U]** or by using any of the **VidIf** output routines.

A single function is provided for you to write trace output to this window. This is the **Trace** procedure, which is a formatted output routine similar to the **printf** function found in C systems. The procedure definition for **Trace** is as follows:


```
PROCEDURE Trace( OutputPat      : ARRAY OF CHAR ;
                  P1,P2,P3,P4    : ARRAY OF BYTE ) ;
```

OutputPat is the "pattern" string that determines the format of output. **P1 – P4** are parameters that will "fill in" fields specified in the pattern string. The simplest form of **Trace** is when it is used to write just a simple string — for example,

```
Trace('Entered procedure',0,0,0,0) ;
```

In this case there are no fields, so the string 'Entered procedure' is output without modification, and **P1 – P4** are ignored.

To enter fields into the string the sequence **%C** is embedded into the string, where **C** is a single character type specifier which determines how the parameter is to be interpreted.

Available types include:

%u — Unsigned	CARDINAL, LONGCARD, SHORTCARD
%i — Integer	INTEGER, LONGINT, SHORTINT
%r — Real	LONGREAL, REAL
%c — Char	CHAR
%s — String	ARRAY OF CHAR
%h — Hex	Unsigned output in hexadecimal format
%b — Boolean	BOOLEAN (TRUE/FALSE)
%p — Pointer	ADDRESS/POINTER

The pattern is scanned from left to right. Whenever a field is encountered, the next parameter is taken from the parameter list (**P1 – P4**) and is expanded into the output string in the format specified.

For example to output two variables **i** and **j**, where **i** is an **INTEGER** and **j** a **CARDINAL**, the following call could be used:

```
i := -1 ; j := 100 ;
Trace(' i=%i, j=%u.', i, j, 0, 0) ;
```

this will print out

```
i=-1, j=100.
```

By default, output is right justified, and is written in as wide a field as needed to display the value. You can override both of these defaults. You can specify the field width to be used for the output by putting an integer between the **%** and the type character. The magnitude of the integer represents the desired field width. Positive

numbers indicate right justification and negative values indicate left justification. For example,

```
Trace(' i=%-4i, j=%5u.', i, j, 0, 0) ;
```

would print

```
i=-1 , j= 100.
```

i is left justified and 4 characters wide; *j* is right justified and 5 characters wide.

For **REAL** values, you can specify both the field width and the number of decimal places to display. To specify the number of decimal places, you need to include an integer preceded by a decimal point between the % and the type specifier. For example,

```
%7.3r
```

specifies a **REAL** to be displayed in 7 columns, with three decimal digits.

In addition to field specifiers, escape sequences may be embedded in the pattern string. These will be replaced by special characters on output. Escape sequences that can be used include the following:

<code>\n</code>	— Newline	write a CR/LF — that is, move to a new line in the output
<code>\f</code>	— Formfeed	clear window
<code>\%</code>	— %	used for output of command char ' % '
<code>\\</code>	— \	used for output of command char ' \ '
<code>\num</code>	— char	Character with ASCII value num

Of these the `\n` is the most often used as it can be used to spread output onto several lines. For example,

```
Trace('File name = %s\nFile size = %u\nFile open = %b', fn, fs, isopen, 0);
```

could output

```
File name = TEST.DAT
File size = 262144
File open = TRUE
```

You can display a % in your output by including either '`\%`' or '`%%`' in your format string.

The full syntax for the pattern string is as follows:

- `FormatString = { Alpha | FieldSpecifier | SwitchChar }`
- `Alpha = any ASCII char except '%' and '\'`

- **FieldSpecifier** = '%' '%'
| '%' ['-'] [WidthSpecifier] TypeSpecifier
- **WidthSpecifier** = DecimalNumber ['.' DecimalNumber]
- **TypeSpecifier** = 'u' (* Unsigned *)
| 'i' (* Signed *)
| 'r' (* Real *)
| 'c' (* Character *)
| 's' (* String *)
| 'h' (* Hex (unsigned) *)
| 'b' (* Boolean *)
| 'p' (* Pointer / Address *)
- **SwitchChar** = '\' SwitchOptions
- **SwitchOptions** = '\' (* \ *)
| '%' (* % *)
| 'b' (* BS = CHR(8) *)
| 'f' (* FF = CHR(12) *)
| 'n' (* NL = CHR(13),CHR(10) *)
| 't' (* Tab = CHR(9) *)
| 'e' (* Esc = CHR(27) *)
| CharCode
- **CharCode** = DecimalNumber
- **DecimalNumber** = Digit [Digit [Digit]]

Display Output Control

In addition to being able to output your information using **Trace**, you can specify the position of the output in the User Trace window. To do this, use the **VidIf** procedure **GotoXY**.

The definition for this procedure is:


```
PROCEDURE GotoXY ( X,Y : CARDINAL ) ;
```

Where **X** and **Y** are the required new coordinates at which subsequent trace output will be displayed. The coordinates are based from 1, so the top left hand corner is 1,1.

To get the current output position you can use the routine **WhereXY**. This has the definition

```
PROCEDURE WhereXY ( VAR X,Y : CARDINAL ) ;
```

and returns the **X**, **Y** coordinates of the position where the next output will be written.

You can use procedure **SetColor** to change the foreground and background of text output. This procedure is defined as follows:

```
TYPE
  Color = ( Black,      Blue,      Green,      Cyan,
            Red,        Magenta,   Brown,     LightGray,
            DarkGray,   LightBlue,  LightGreen, LightCyan,
            LightRed,   LightMagenta, Yellow,     White );
```

```
PROCEDURE SetColor ( Fore,Back : Color ) ;
```

The parameters **Fore** and **Back** set the colors of the text foreground and the text background, respectively. This will affect all subsequent output and can be used to highlight information selectively. Setting a high background (**DarkGray .. White**) may cause text to "blink" on a CGA adaptor.

Use the procedure **InitDebugWindow** to set the size of the output window and the title displayed. This is defined as:

```
PROCEDURE InitDebugWindow ( Title      : ARRAY OF CHAR ;
                           Width,Depth : CARDINAL ) ;
```

For example, the call

```
InitDebugWindow('Sample Data',20,5) ;
```

will set the user trace window to 20 characters across and 5 lines deep with the heading 'Sample Data'. This procedure will also automatically open the User Trace window if this has not already been opened. You can open and close this window from your program by using the following routines:

```
PROCEDURE OpenDebugWindow ;
```

```
PROCEDURE CloseDebugWindow ;
```

OpenDebugWindow opens the User Trace window and puts it on top of the window stack while **CloseDebugWindow** removes the User Trace window from the screen. In such a case, the window still exists, even though it is inactivated. This means that output (by **Trace**) will still be written to the window even when it is not visible. This material can be read when the window is visible again.


```

VAR
  ha      : CARDINAL ;
BEGIN
  ha := Storage.HeapAvail(Storage.MainHeap) DIV 64 ;
  VidIf.Trace(' Storage available at %10s(%u) = %3u Kbytes\n',
              modname,lineno,ha,0) ;
  RETURN (ha<4);          (* stop if less than 4K left *)
END StoreTraceTrap;

BEGIN
  (* Connect VID User breakpoint to StoreTraceTrap *)
  VidIf.SetUserTrapProc(StoreTraceTrap) ;
  ...

```

When debugging, you can install user breakpoints at any points in your program where you want or need to check the storage. See the example program **StorTest** for a more fully implemented example including full storage consistency checks.

To disconnect the user breakpoint from the procedure you've specified, you can set up another user trap handler using **SetUserTrapProc** or you can call the procedure **ResetUserTrapProc**. This procedure stops the calling of any user trap handlers.

You can also remove the User Breakpoint that invoked the procedure by calling the procedure **ClearUserTrap**. This can *only* be called from within the User Breakpoint procedure and will only remove the specific breakpoint that caused the procedure to be called.

Debugging Multiple Processes

The Modula-2 language supports coroutines via **TRANSFER** and **IOTRANSFER**, and the TopSpeed Modula-2 system provides a **Process** module that implements a time-slicing process scheduler. These allow the creation of programs with more than one process.

Debugging such multiple-process programs can sometimes prove difficult as the flow of execution is often more complicated to follow than in a normal single process.

VID does not provide specific support for different scheduling systems. Rather, it provides a single option that, when enabled, ensures that *only* the current process is traced or trapped by VID. Thus when debugging multiple processes you can isolate the specific process that you are interested in and follow its execution. This option is invoked by Options single Process ( ) and may be saved using the normal option save mechanisms.

When using a time-slicing schedule that is based on the PC timer interrupt (such as the TopSpeed **Process** module) you should note that VID will "freeze" all processes executing whenever it regains control. This is to prevent executing processes from interfering with VID's execution.

To illustrate how a multiple-process module can be debugged, the demo program **GenPrime** is provided. This program generates a sequence of large, 8-digit primes. The program is implemented as 4 processes — plus the main program. (The program implements a rather obscure and inefficient method invented especially for this example.)

The four processes perform the following functions:

TrialProcess	This process generates random numbers in the correct range and then loops looking for factors of the number.
TimeoutProcess	This process keeps “watch” on the TrialProcess . If the TrialProcess takes more than 5 seconds, then its search for factors is terminated and the trial number is passed on to the checking process to be validated.
CheckProcess	This waits for possible prime candidates and then checks that the number is prime. If it is then the number is printed.
StatusProcess	If Scroll Lock is pressed, this process will monitor and print out the number currently being tested by the TrialProcess .

The main program terminates after 60 seconds of execution time.

Debugging a Multiple-Process Program

To debug this program, compile and link it using the required options. Then type

```
VID GenPrime
```

With the **Option single Process** set to OFF, you can trace through all the processes as the program executes. The execution may appear to “jump” from line to line — either because of implicit **TRANSFERING** (that is, through procedures such as **StartProcess**, **SEND**, **Delay**, etc.) or because of the time-slicing of processes.

To see the movement between processes, start executing **GenPrime**. Open the Procedure Trace window (**T** **P**). You'll find **GenPrime** as the main active procedure. Single step to the call

```
Process.StartProcess(TrialProcess ,2000H,1);
```

When you press **Space** again, the procedure **TrialProcess** will be activated and your tracing will continue in this process. You'll notice this fact in the Procedure Trace window, since the “top” procedure has changed to **GenPrime.TrialProcess**.

Debugging Multiple Processes One at a Time

Suppose that you only wish to debug, say, **CheckProcess**. To do this, use **[B]** **[P]** or **[+]** to set a breakpoint in the “root” procedure **CheckProcess** — on the first executable line of this procedure. Then type **[G]** **[B]** (Go Breakpoint) and let processing continue.

After a few seconds, the trap is hit and you have reached the **CheckProcess** (Check this by using the Trace Procedure command). Now if you enable the **Option** single **Process** (**[O]** **[P]**), you will isolate the debugging to this particular process.

Tracing program execution — continuous or single step — will only display this process and breakpoints will be recognized only within this process. For example try **[G]** **[T]** (Go trace) or set a breakpoint in the **WrMsg** procedure (which is also called from **TimeOutProcess** and **StatusProcess**). This breakpoint will only be triggered when **WrMsg** is called from **CheckProcess**.

To re-enable multiple-process operation, simply set the **Option** single **Process** switch back to OFF. This ability to treat processes individually or together allows you to use all the features available under VID to debug complicated, multiple-process programs with ease.

Modula-2 Utilities

Your VID package also includes some utilities that you may find useful for particular purposes. These utilities are described in the following sections.

M2DASM

This program provides a means of generating a symbolic disassembly listing from a program **EXE** file. The disassembly process generates a text file with the default extension **.DAL**.

Because of the size of disassembly output you can specify exactly which segments to disassemble. To do this you can either specify a list of segment names (with wildcards ***** and **?** allowed) or you can select the segments interactively (by not specifying a segment list when you call the program).

The syntax for calling **M2DASM** is as follows:

```
M2DASM [/option] progname {segments}
```

The following options are allowed for **M2DASM**:

- /O <name>** This specifies **name** as the output file for the program. If this is not specified, then the default filename, **programe.DAL** is used.
- /L** This option tells **M2DASM** to display source line numbers interleaved with the assembler listing.
- /S** This option tells **M2DASM** to display the Modula-2 source text interleaved with the assembler listing.
- /H** This tells **M2DASM** to display hex bytes as well as the symbolic opcodes for disassembled output.
- /D** This tells **M2DASM** to display the contents of initialized data segments in hexadecimal and ASCII formats.

The following are example uses of the **M2DASM** program:

M2DASM VidDemo

This generates **VidDemo.DAL** after prompting for segments.

M2DASM /l/s/h/d VidDemo

This generates a full mixed source/disassembly listing with line numbers and hex bytes.

M2DASM /oviddemo.lst viddemo

This writes the program output to **VidDemo.lst**.

M2DASM /lshd VidDemo C_VidDemo C_PushOver

This generates code listings for segments **C_VidDemo** and **C_PushOver** without prompting.

M2DASM /lsh VidDemo C_*

This disassembles all code statements (**C_***) without prompting.

Note that to disassemble an **EXE** file, the map file for the program must be present. When using the **/L** and **/S** options, line number information should be generated by the *compiler*. Also for the **/S** option the Modula-2 source should be locatable, either in the current directory or through the redirection file.

Here is an example disassembly generated using the command line

```
m2dasm /lsh sieve c_sieve
```

This produces the following information:

```
SEGMENT C_Sieve                ; Class = FCODE           Module = Sieve
                                ; Start = [006D:000A]       Length = 92
```

```
=====
sieve.MOD
=====
```


Line: 12 - WrStr('Sieve, '); WrCard(NoOfIterations,0); WrStr(' iterations');

Sieve@

```
000A: B8 08 00      MOV     AX,8
000D: 50              PUSH    AX
000E: 0E              PUSH    CS
000F: B8 9E 00      MOV     AX,9E
0012: 50              PUSH    AX
0013: 9A 50 00 78 00 CALL    IO$WrStr
0018: B8 FA 00      MOV     AX,0FA
001B: 50              PUSH    AX
001C: 2B C0          SUB     AX,AX
001E: 50              PUSH    AX
001F: 9A A7 01 78 00 CALL    IO$WrCard
0024: B8 0C 00      MOV     AX,0C
0027: 50              PUSH    AX
0028: 0E              PUSH    CS
0029: B8 A6 00      MOV     AX,0A6
002C: 50              PUSH    AX
002D: 9A 50 00 78 00 CALL    IO$WrStr
```

Line: 13 - FOR iter:=1 TO 250 DO

```
0032: 2E 8E 1E 9C 00 MOV     DS,CS:Word Ptr[009C]
0037: B9 01 00      MOV     CX,1
```

Line: 14 - count:=0;

```
003A: 2B F6          SUB     SI,SI
```

Line: 15 ; FOR i:=0 TO Size DO Flags[i]:=TRUE

```
003C: 2B DB          SUB     BX,BX
003E: C6 87 00 00 01 MOV     Byte Ptr[BX+0],1
0043: 43             INC     BX
0044: 81 FB FE 1F    CMP     BX,+1FFE
0048: 76 F4          JBE     003E
```

Line: 17 ; FOR i:=0 TO Size DO

```
004A: 2B DB          SUB     BX,BX
```

Line: 18 ; IF Flags[i] THEN

```
004C: 80 BF 00 00 00 CMP     Byte Ptr[BX+0],+0
0051: 74 1D          JE      0070
```

Line: 19 - prime:=i*2+3;

```
0053: BA 03 00      MOV     DX,3
0056: 89 D8          MOV     AX,BX
0058: D1 E0          SHL     AX,1
005A: 01 C2          ADD     DX,AX
```

Line: 20 - k:=i+prime;

```
005C: 89 D7          MOV     DI,DX
005E: 01 DF          ADD     DI,BX
```

Line: 21 - WHILE k<=Size DO

```
0060: 81 FF FE 1F    CMP     DI,+1FFE
0064: 77 09          JA      006F
```



```

Line: 22      -      Flags[k]:=FALSE;
0066: C6 85 00 00 00      MOV      Byte Ptr[DI+0],0
Line: 23      -      INC (k,prime);
006B: 01 D7      ADD      DI,DX
006D: EB F1      JMP      0060
Line: 25      -      INC (count);
006F: 46      INC      SI
0070: 43      INC      BX
0071: 81 FB FE 1F      CMP      BX,+1FFE
0075: 76 D5      JBE      004C
0077: 41      INC      CX
0078: 81 F9 FA 00      CMP      CX,+0FA
007C: 76 BC      JBE      003A
Line: 29      -      WrLn; WrCard (count,6); WrStr (' primes');
007E: 9A EF 01 78 00      CALL     IO$WrLn
0083: 56      PUSH     SI
0084: B8 06 00      MOV      AX,6
0087: 50      PUSH     AX
0088: 9A A7 01 78 00      CALL     IO$WrCard
008D: B8 08 00      MOV      AX,8
0090: 50      PUSH     AX
0091: 0E      PUSH     CS
0092: B8 B2 00      MOV      AX,0B2
0095: 50      PUSH     AX
0096: 9A 50 00 78 00      CALL     IO$WrStr
Line: 30      -      END Sieve.
009B: CB      RETF
END SEGMENT Sieve

```

M2PROF

This utility enables you to make an execution profile of your program as it runs. This allows you to find out exactly where the program is spending its time. Once you have this information, you can concentrate your optimization energy on the places where these efforts can have the greatest effect.

The execution profile runs your program after generating a symbol table by reading the map. Part of **M2PROF** stays resident while your program runs and samples your program at random intervals (about 1000 times a second). When your program finishes executing, a text file is generated showing the number of times each location was sampled.

You can display the sampling information in either of two modes. The first (default) is by procedure name and the second is by line number. The second mode requires the **/L** option. You can only use the second mode on modules for which line numbers have been generated.

If **M2PROF** cannot recognize the location it is sampling, it will display the range sampled followed by "Unknown." If the information is outside your program, **M2PROF** will print the owning program's name.

The syntax for calling **M2PROF** is as follows:

```
M2PROF [/option] progname [proparams]
```

You also have the following options available, for use with **M2PROF**:

- /O <name>** This sets the output file to the name specified. If no name is specified, then the default name, **progname.PRF** is used.
- /L** This generates the profile based on source line numbers. If this is not set then the profile will be based on procedures. To use the **/L** option you must have line number information generated for modules you want included in the profile.

Here is an example call using **m2prof**

```
m2prof /l sieve
```

The following shows a sample execution profile:

Execution Profile for SIEVE.EXE

Program base at segment 39A8

Seg : Range	Symbol name	Samples
0170:14D6-5F8F	Unknown (DOS)	22
0C02:011B-031E	Unknown ()	6
107B:89AE-98A0	Unknown (KEY.COM)	4
1FF2:2DD4-3AF8	Unknown (SK2E.EXE)	11
2E5D:006B-00A0	Unknown (SK2E.EXE)	3
39A8:04AD-04AD	Unknown	1
3A15:003A-0049	sieve.MOD Line: 15	12289
3A15:004A-0052	sieve.MOD Line: 18	10358
3A15:0053-005B	sieve.MOD Line: 19	1229
3A15:005C-005F	sieve.MOD Line: 20	577
3A15:0060-0065	sieve.MOD Line: 21	14421
3A15:0066-006A	sieve.MOD Line: 22	5639
3A15:006B-006E	sieve.MOD Line: 23	5777
3A15:006F-007D	sieve.MOD Line: 25	8979
3A20:02C4-02C4	Unknown	1
3A54:00DA-00DA	Unknown	1
F000:DA46-F087	Unknown (BIOS)	18

M2SIZE

This utility prints a sorted list of procedures and their sizes, in descending order. It also prints the information about code present for each module, together with that module's percentage of the total code in the program.

The syntax for **M2SIZE** is

```
m2size [/option] <name>
```

The only option available with **M2SIZE** is

/O <name> This lets you specify where to write output information. If this is not specified, then the default **programe.SIZ** is used.

TechKit™

The TopSpeed Modula-2 TechKit offers an abundance of useful technical information, utilities, and program modules.

Write communications programs.

Asynchronous communications are made easy by the RS module supplied in the TechKit. RS is an interrupt-driven communications driver written entirely in Modula-2. It features easy installation and buffered I/O. Full source supplied together with a demonstration communications program.

Write assembler routines using TopSpeed Assembler.

TopSpeed Asm assembles 30,000 lines/minute and has been designed especially for TopSpeed Modula-2. Unlike other assemblers, TopSpeed Asm generates library files directly and is, therefore, superbly suited for use with the TopSpeed smart linker. TechKit contains complete documentation on the TopSpeed Assembler.

Write pop-up Terminate and Stay Resident programs a la Sidekick.

Using the module TSR supplied with Techkit, it is a breeze to write a pop-up application. Simply place all of your program in a single procedure and pass the procedure to TSR together with your memory requirement and how you want your program to be activated.

Implement a critical error handler.

DOS interrupt 24H is the critical error handler that is called when a hardware error occurs (such as an open disk drive door, a printer that is off-line, etc.). The DOS version of this interrupt handler usually displays: Abort, Retry, Ignore? With the TechKit's critical error handler, you can avoid this by taking control of this interrupt.

Use TopSpeed Modula-2 for imbedded systems.

TechKit contains all the necessary information to make ROMable code. You receive a ROMable version of the run-time library in TopSpeed Assembler, ready to modify for your specific requirements. Also, you will receive a program that enables you to specify exactly where the various segments of your program are to be placed.

Make use of Lotus/Intel/Microsoft expanded memory.

TechKit contains a complete set of procedures to interface to the Expanded Memory Manager (EMM).

Complete assembler source for start-up code.

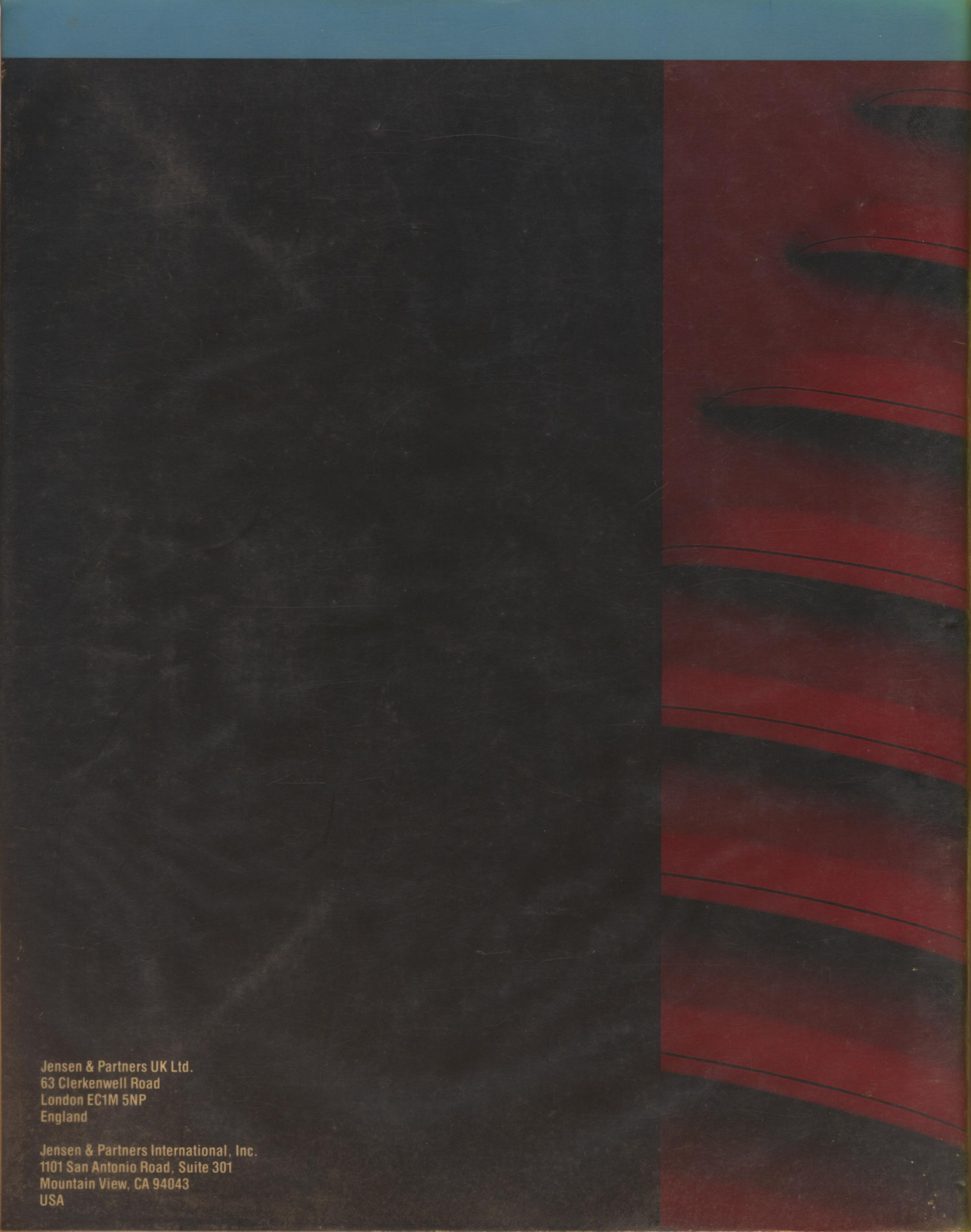
To give you complete control of your program's final code, the TechKit contains the start-up code and run-time code, which are added to all programs automatically.

To order, mail in the coupon on the next page, or call:

1-800-543-5202 in the U.S.

1-800-543-8452 in Canada

01-253-4333 in England



Jensen & Partners UK Ltd.
63 Clerkenwell Road
London EC1M 5NP
England

Jensen & Partners International, Inc.
1101 San Antonio Road, Suite 301
Mountain View, CA 94043
USA