

Tbug User Guide

Tbug User Guide

3L Ltd



Copyright © 2001 by 3L Ltd. All Rights Reserved.

This edition April 17, 2001 describes version 1.0.21 of the software.

20	19	18	17	16	15	14	13	12	11
10	9	8	7	6	5	4	3	2	1

3L[®] is a registered trademark, and the 3L logo is a trademark of 3L Ltd.

3L Ltd
86/92 Causewayside
Edinburgh EH9 1PY
Scotland
Tel. +44 131 662 4333
Fax. +44 131 662 4556
E-mail Support@ThreeL.Co.UK

Contents

Introduction	xi
Intended Audience	xi
Hardware Assumptions	xi
Processor Types	xii
Software Assumptions	xii
Associated Documents	xii
Conventions	xiii
1 Installation	1
2 Overview	5
2.1 The Window System	5
2.2 Architecture	6
2.2.1 Configured and Non-Configured Applications	6
2.2.2 Working on a Thread	7
2.2.3 The Console Window	8
2.3 Preferences	8
2.4 Unexpected Results	8
3 Worked Example	11
4 Compiling and Linking	51
4.1 Compilation	52
4.2 Linking	52
5 Starting Tbug	55

5.1	Option Switches	57
5.2	Configured Applications	57
6	The Window System	59
6.1	Basic Window Operations	60
6.1.1	Selecting Windows	60
6.1.2	Overlapping Windows	60
6.1.3	Moving Windows	61
6.2	Ordinary Windows	61
6.2.1	The Initial Display	62
6.2.2	Creating Ordinary Windows	62
6.2.3	Deleting Ordinary Windows	63
6.2.4	Thread Windows	63
6.3	Help Windows	65
6.4	Special Windows	66
6.4.1	Alerts	67
6.4.2	Edit Boxes	68
6.5	Global Commands	69
6.5.1	Display Version Command	69
6.5.2	Exit Command	69
6.5.3	Other Global Commands	70
7	Source-Level Features	71
7.1	Availability	71
7.2	Source Windows	72
7.2.1	Cursor Bar	72
7.2.2	Execution Marker	74
7.2.3	Single-Stepping	74
7.2.4	Execute to Here	75
7.2.5	Free-Running	75
7.2.6	Error Conditions	76
7.2.7	Break-in	77
7.2.8	Breakpoints	78
7.3	Variables Windows	79
7.3.1	Observing Variables	79
7.3.2	Changing the Values of Variables	83
7.3.3	Information about Variables	84

7.3.4	Changing the Type of Variables	84
7.3.5	Arrays	90
7.3.6	Opening a Memory Window on a Variable . . .	92
7.4	File Windows	93
8	Low-Level Features	95
8.1	Assembler Windows	95
8.1.1	Locking	96
8.1.2	Single-Stepping	97
8.1.3	Skipping an Instruction	97
8.1.4	Breakpoints	97
8.1.5	Execute to Here	97
8.1.6	Move to Here	98
8.1.7	Following Jumps and Calls	98
8.1.8	Backtracking	98
8.1.9	Moving Assembler Windows	98
8.1.10	Duplicating Assembler Windows	101
8.2	Register Windows	101
8.2.1	Integer Registers	102
8.2.2	Floating-point Registers	103
8.3	Memory Windows	105
8.3.1	Creating Memory Windows	105
8.3.2	Locking	106
8.3.3	Changing the Display Format	107
8.3.4	Changing the Display Location	108
8.3.5	Duplicating Memory Windows	110
8.3.6	Changing Memory Values	110
9	Parallel Execution	113
9.1	Multiple Threads	113
9.2	Multiple Tasks	114
9.2.1	Starting Configured Applications	114
9.2.2	Working with Tasks	115
9.2.3	Flood-Filled Configuration	117
9.2.4	General Configuration	118
9.2.5	Choice of Configuration	118

10 Memory and Link Protection	121
10.1 The Master Control Panel	122
10.2 The Thread Control Panel	123
10.3 Disabling Memory Protection	124
10.4 Disabling Link Protection	124
11 Environment	127
11.1 Persistent Data	127
11.2 WinEdit	128
11.3 Preferences	129
Appendices	135
A Command Summaries	135
A.1 General List of Commands	135
A.2 Classified List of Commands	138
A.2.1 Controlling Windows	138
A.2.2 Cursor Control	139
A.2.3 Commands for Special Windows	139
A.2.4 Commands for Help	140
A.2.5 Global Commands	140
A.2.6 Execution and Breakpoint Commands	141
A.2.7 Commands for Source Windows	141
A.2.8 Commands for Variables Windows	142
A.2.9 Commands for Array Windows	142
A.2.10 Commands for File Windows	143
A.2.11 Commands for Assembler Windows	143
A.2.12 Commands for Memory Windows	144
A.2.13 Commands for the Task Directory	144
A.2.14 Commands for Thread Directories	145
A.2.15 Commands for Console Window	145
A.2.16 Commands for Edit Boxes	145
B NEC PC-9801 Hosts	147
B.1 Installation	147
B.2 The NEC PC-9801 Keyboard	148

B.3 The NEC PC-9801 Display	149
C Distribution Kit	151
Bibliography	153
Index	155

Introduction

Intended Audience

This *User Guide* accompanies 3L's Tbug product and is intended for use by anyone who wants to develop programs on transputer systems using 3L's Parallel compilers. We assume that you are familiar with Parallel C and the terminology used in the *Parallel C User Guide*[3].

Hardware Assumptions

You can use Tbug on a variety of transputer systems. This guide makes the simplifying assumption that your hardware is an Inmos IMS B004 transputer evaluation board, or a transputer system which is largely compatible with a B004. This board is a single plug-in card for the standard IBM PC bus, with one transputer and either 1MB or 2MB of RAM.

Similarly, the assumption is made here that the host computer for the B004 will be an IBM PC with a hard disk drive, or one of the many personal computers compatible with the original IBM machines. Tbug supports all standard IBM text video modes.

Processor Types

Running on a T414 transputer, Tbug can only process programs compiled for the T414.

On the T800, Tbug can process programs compiled for the T414, provided they do not use the T414's special single-precision floating point instructions. In the case of programs written in Parallel C, this will only happen if you use `float` variables and the program is compiled with the `/S` switch. See the *Parallel C User Guide*[3], Appendix D.

Software Assumptions

Tbug assumes that it will be run under the MS-DOS operating system.

The full range of facilities described in this User Guide is available with programs developed using 3L Parallel C version 2.1 or later. In addition, Tbug will handle programs developed using version 2.0 of any of the 3L Parallel compilers. In this case Tbug will not be able to identify internal variables by their names, but in other respects will function as described in the User Guide.

Tbug should be used in conjunction with version 1.0 or later of the 3L Linker, as only this linker can build the tables of information which Tbug needs to work at the source level. Programs linked with older linkers may be debugged, but in this case the facilities described in chapter 7 will not be available.

Associated Documents

Information about compiling C programs, configuration and much else may be found in the *Parallel C User Guide*[3].

Information about linking programs may be found in the *Linker User Guide*[6].

Information about MS-DOS, its file system and command structure may be found in *Microsoft MS-DOS User's Reference*[2,1].

Users of Parallel Fortran and Parallel Pascal should also have to hand the *User Guide* for their compiler[4,5].

Conventions

Throughout this manual, text **printed in this typeface** represents direct verbatim communication with the computer; for example, pieces of C text, commands to MS-DOS and responses from the computer.

In examples, text *printed in this typeface* is not to be used verbatim; it represents a class of items, one of which should be used. For example, a command used to invoke Tbug could be described as:

tbug *filename*

This means that the actual input will consist of:

1. the word “**tbug**” typed exactly like that;
2. a space; and
3. a *filename*: not the word “*filename*” but something which the accompanying description explains, in this case the name of a file, such as **prog.b4**.

Chapter 1

Installation

This chapter contains instructions on how to copy Tbug from the floppy disks supplied onto your hard disk, and make it ready for use. You can skip this chapter if the correct version of Tbug has already been installed on the machine you are using. All users of the NEC PC-9801, however, should read appendix B before reading the rest of this manual.

We have assumed that your floppy disk is drive A: and that your hard disk is drive C:. If this is not the case all you have to do is substitute the correct drive letters in the installation commands.

Tbug is distributed on two $5\frac{1}{4}$ " 360KB floppy disks. Appendix C gives a list of the files to be found on these disks. Disk 1 contains the files for Tbug itself while disk 2 contains some sample programs, including the one used in the worked example in chapter 3 of this User Guide. All of the screen layouts in this guide have been generated from these sample programs.

You install Tbug by copying the files from floppy disk 1 to a directory on your hard disk. There are three possibilities when you are choosing a directory for this.

1. A directory where you keep MS-DOS commands and other commonly used programs for general use. This might be the top-level directory `c:\`, or a subdirectory. To install Tbug into the top directory, you should use the following command:

```
C>copy a:*. * \
```

In the case of a subdirectory, you might use a command like this:

```
C>copy a:*. * \dos
```

2. The directory with the 3L compiler you use (for example, the Parallel C directory, `c:\tc2v1`). To install Tbug into this directory give the command:

```
C>copy a:*. * \tc2v1
```

3. A directory created specially to hold Tbug (for example, `c:\tbug1v0`). To install Tbug into this directory give the commands:

```
C>md \tbug1v0  
C>copy a:*. * \tbug1v0
```

Once these operations have been successfully completed you will be able to access Tbug from the directory into which you have copied it. Before you can use it elsewhere you must make sure that the directory containing Tbug is on the MS-DOS *search path*. Program files stored in directories which are on the search path can be loaded and run simply by typing the name of the program as a command. In addition, Tbug requires access to its persistent data files (see chapter 11) and will not be able to find them unless the search path has been set up correctly.

The search path for your machine is set up by commands in the batch file `\autoexec.bat` which are automatically executed when your machine is started up. To change the search path you will need

to edit `autoexec.bat` using your text editor. The `autoexec.bat` file will probably already contain a line of the following form:

```
path directory; directory;...
```

For example:

```
path c:\dos;c:\utils;c:\tc2v1
```

If this path statement already refers to the directory into which you have installed Tbug then you do not need to make any alterations. Otherwise, simply add the full name of the directory to the end of the `path` statement, using a semicolon to separate the extra text from the text already there. For example, if you have installed Tbug in a directory called `c:\tbug1v0` then the path command given in the previous example would be changed to:

```
path c:\dos;c:\utils;c:\tc2v1;c:\tbug1v0
```

The current version of Tbug restricts the total length of the search path to 100 characters, i.e., somewhat more than a line of text. If adding a new directory would cause this limit to be exceeded, you should either change the `path` command by removing one or more of the other directories or install Tbug into a directory already in the path.

If you have had to alter the `path` statement you should re-boot your system so that the new path will be set up.

When you have completed this installation procedure you will be able to access Tbug from any directory on your system. However, as we noted in the Introduction above, to use all the functions of Tbug, you will need version 1.0 or later of the 3L Linker, so this should be installed before you start to use Tbug. Instructions for installing the 3L Linker can be found in *3L Linker User Guide*[6].

Installing the software should be trouble-free. If you do encounter any problems, please repeat the procedure outlined in this chapter carefully. If this does not clear the problem up, please contact your supplier for further help.

Chapter 2

Overview

Tbug is an interactive, window-based, source-level debugger, which has a wide range of facilities. These include:

- Debugging programs made up of multiple tasks and threads.
- Controlling the program by starting, stopping and interrupting execution.
- Observing the flow of control as the program executes.
- Observing the values of variables and memory locations as the program executes, and changing them if required.

These facilities are provided through a number of *windows* displayed on your screen. You control the program being debugged by interacting with these windows.

2.1 The Window System

The window system of Tbug is described in detail in chapter 6.

At any time, there may be one or more windows being displayed. One of these is *selected*, and this is indicated by some kind of highlighting. It is only this selected window that you can communicate with or manipulate. You can select another window by pressing F5; each time you do this, a different window will be selected, until you get back to the window you started from.

Sometimes, the window system will display a window which needs to be dealt with at once: it could be an error report, a request for data input or one of several different kinds of menu. You will have to attend to this window, which will be highlighted in a special way, before you can do anything else.

The window system includes an extensive library of *help displays*. To get help about a particular window, select it and press the F1 key.

2.2 Architecture

2.2.1 Configured and Non-Configured Applications

Tbug can debug two kinds of application, which are treated slightly differently. They are:

- multiple-task *configured* applications produced by one of the 3L configurers, `config` and `fconfig`;
- single-task *non-configured* applications, produced by the linker directly.

These two types of application are loaded for debugging by Tbug in different ways. Refer to chapters 5 and 9 for more information about this.

A configured application is always debugged in one transputer, even if the configuration specified that several transputers should be used.

When a configured application is debugged, Tbug displays a window called a *task directory*. This contains a list of all the tasks in the application, and by using it, you can select tasks to work on. When you select a task, that task's *thread directory* is displayed. This contains a list of all the currently active threads in that task, and enables you to select threads to work on.

With a non-configured application, a task directory is not displayed. Instead, Tbug moves straight to displaying the thread directory for the single task.

Task and thread directories may be deleted from the display by selecting them and pressing the Del key.

2.2.2 Working on a Thread

In order to work on a thread, Tbug provides a number of different kinds of window, which are known collectively as *thread windows*. Each thread can be regarded as having its own stack variables and registers, including its own instruction pointer (**Iptr**) and workspace pointer (**Wptr**). Every thread window is connected to one thread, and the information displayed in the windows connected to a thread reflects the contents of that thread's stack variables and registers.

The facilities supplied by the various thread windows include executing the program, setting breakpoints for the thread, tracing and changing the values of its variables, inspecting files and much else. Thread windows in general are discussed in section 6.2.4, and each type is also examined in later chapters.

Any thread window can be deleted by selecting it and pressing the Del key.

2.2.3 The Console Window

The *console window* is used to display output which the program being debugged sends to the console. In addition, if you select the console window and press the Ins key, you will raise a task or thread directory, depending on whether you are working on a configured or non-configured application. This is useful, as the console window can never be deleted; even if you delete all the other windows, you can build them up again, starting with the console window.

2.3 Preferences

Facilities are supplied for changing the way Tbug works to suit your preferences. This includes things like the ways windows are displayed (colours, borders, etc) and various execution options for the debugger. All this is discussed in chapter 11.

2.4 Unexpected Results

Occasionally, running a program under the control of Tbug may produce results which are different to those produced when the program runs on its own. Some causes of differences are:

1. Certain run-time error conditions which, when running outside Tbug, would not be detected and would result in the program behaving in unpredictable ways, are trapped by Tbug and reported to the user. See section 7.2.6 for details.
2. Timing dependencies. When several independent tasks are being executed it is possible that the exact time taken for functions to be performed can affect the results. Programs with such dependencies are likely to be very unstable and misbehave even under ideal conditions.

3. Address dependencies. When a program is executed under the control of Tbug it is unlikely that it will occupy the same memory addresses as when it runs alone. This can change the behaviour of a program containing assumptions (or errors) relating to actual addresses.
4. Undefined values. A common programming error is to use the contents of a variable before it has been assigned a value. When running without Tbug, the result of this error is that the program gets whatever value had been left in memory previously. However, Tbug preinitialises all memory to a particular value, so that programs will tend in such cases to have reproducible behaviour.

Chapter 3

Worked Example

This chapter gives a worked example of using Tbug on a simple program.

Once you have installed Tbug, as described in chapter 1, we recommend that you should follow through this chapter, using the example program supplied in your distribution kit. To do this, you should create an empty working directory, and copy in the files from distribution disk 2.

For example:

```
C>md worked
C>cd worked
C>copy a:*.*
```

For reference, figure 3.1 is a listing of the example program used in this chapter.

```
main()
{
    int times,first,count,answer;
    first=1;
    count=2;
    answer=3;
    for (times=1; times<=3; times++) {
        first += 1;
        count -= 1;
        answer = fred(first, count);
    }
    *(char *)0x12345678 = 0;    /* very silly */
}

fred(arg1, arg2)
int arg1,arg2;
{
    int temp;
    temp = arg2+arg1;
    if (temp&1) return temp-1;
    else return 2-temp;
}
```

Figure 3.1: Example program for worked example

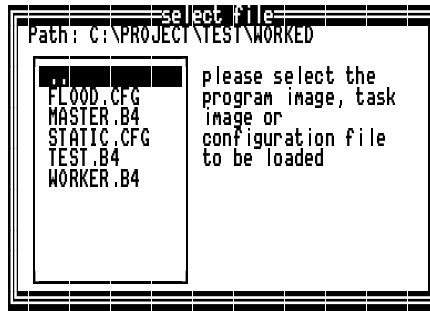


Figure 3.2

Once you have copied the example programs into your working directory, you can invoke Tbug by giving the command:

```
C>tbug
```

This should display on the screen a *program selection menu*, that is, a list of the files and directories which Tbug can deal with, as shown in figure 3.2.

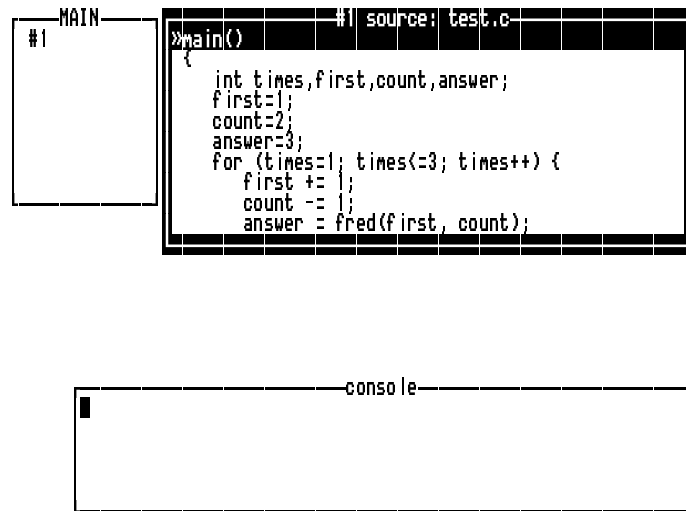


Figure 3.3

Select the test program by pressing ‘↓’ several times, until the cursor bar lies on TEST.B4; then press Enter. This should result in the display shown in figure 3.3. Notice that Tbug has located the correct source file for the test program, TEST.C, and displayed it at the top of the screen.

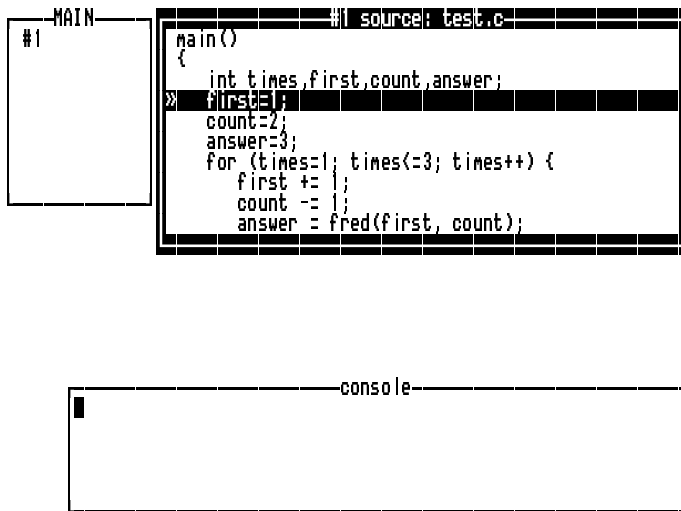


Figure 3.4

If you now press the space bar the program will execute one line and give figure 3.4. Notice that Tbug has skipped over some declarations as these do not result in any instructions to be executed; Tbug has advanced to the next executable statement. This is known as *single-step* execution.

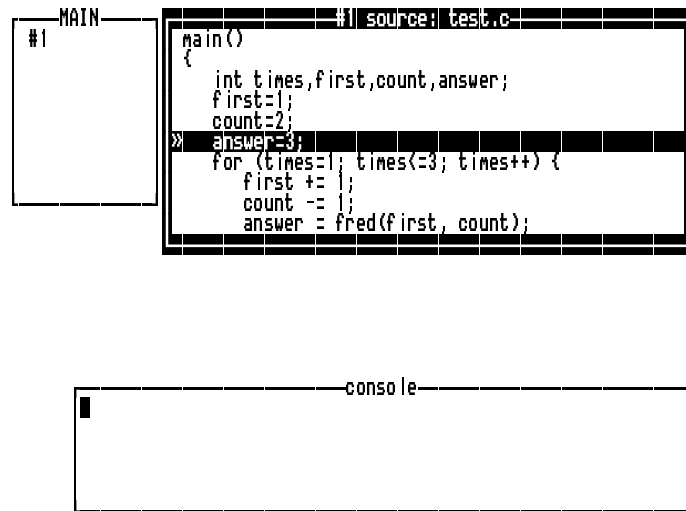


Figure 3.5

Pressing the space bar another twice will step the program through the next two lines, leaving the screen as in figure 3.5.

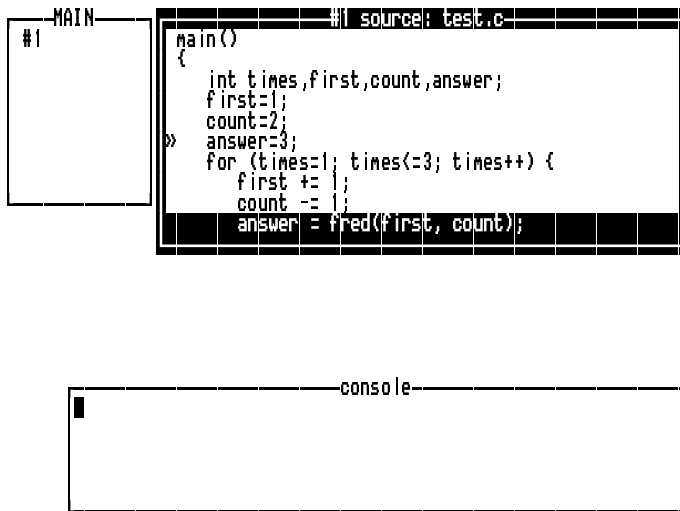


Figure 3.6

You can separate the cursor bar from the execution point marker to the left of the current line by pressing the arrow key '↓' four times. This moves the cursor bar down to the position shown in figure 3.6.

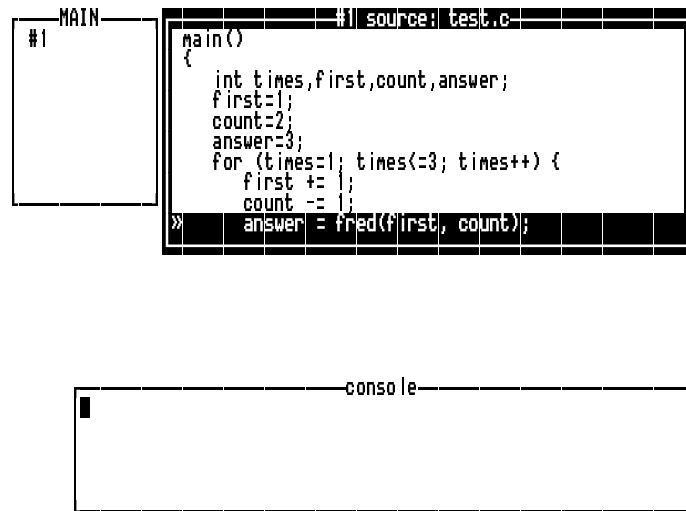


Figure 3.7

Having selected a line with the cursor, you can make Tbug “execute to here” by pressing ‘H’, giving figure 3.7.

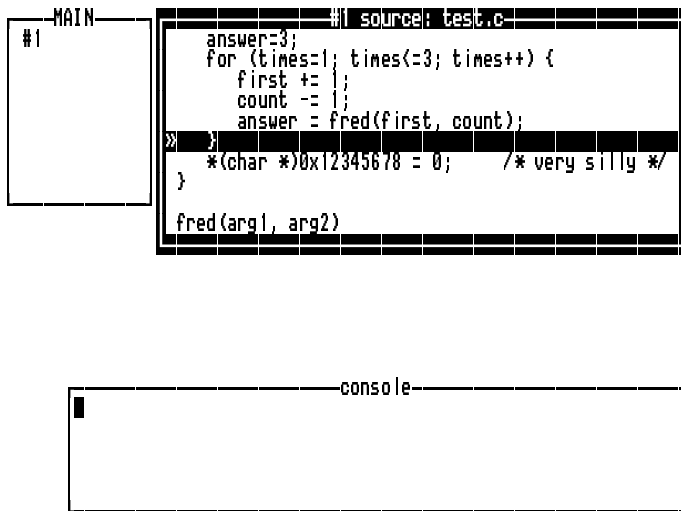


Figure 3.8

You will notice that the current line contains a call to the function `fred`. If you now press the space bar once Tbug will execute the whole of the current line and stop on the following line, as shown in figure 3.8.

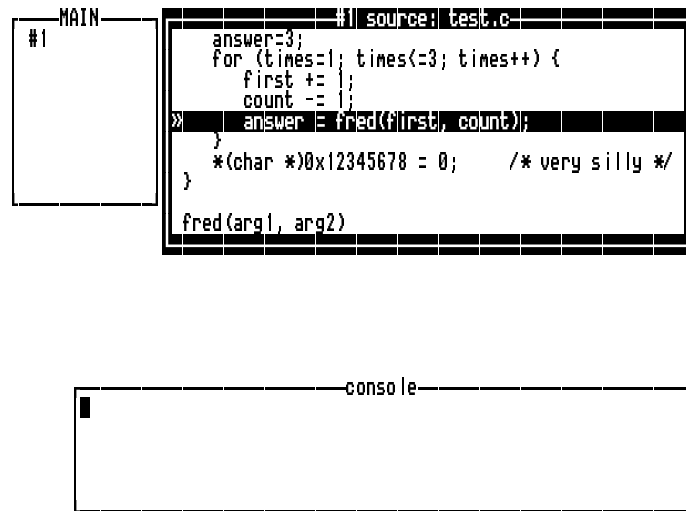


Figure 3.9

Tbug has executed the whole of the call to the function `fred` before stopping. This is known as *stepping over* the function.

If you now type ‘↑’ followed by ‘H’, the program will continue round the loop and stop just before the next call of `fred`, giving the display shown in figure 3.9.

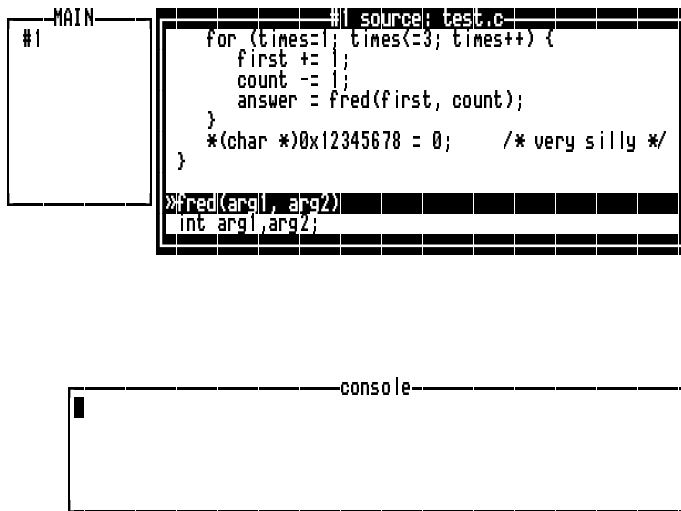


Figure 3.10

This time, instead of typing a space, hit Enter. Now Tbug will *step into* the function and leave the cursor bar on its first statement, as shown in figure 3.10.

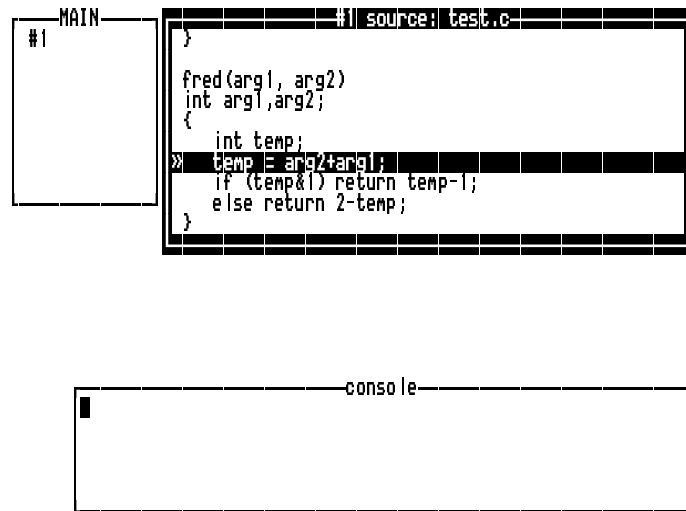


Figure 3.11

Now hit the space bar and Tbug will execute the entry to the function `fred` and leave you on its next instruction, as in figure 3.11.

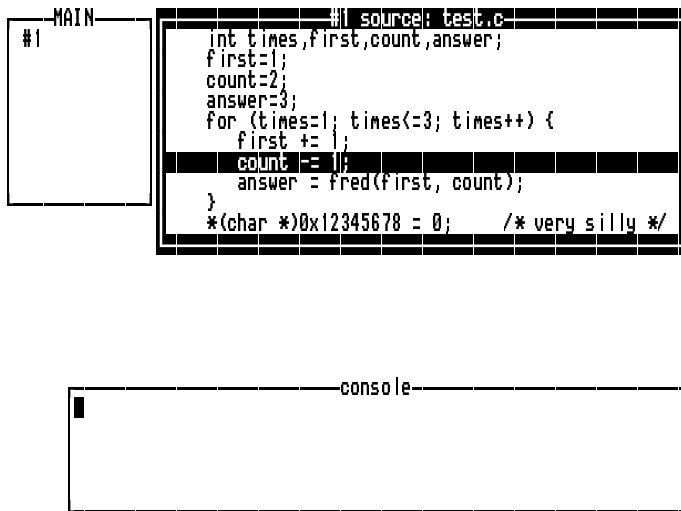


Figure 3.12

To get back to the main program we can use a *breakpoint*. First move the cursor back up the source file until it is on the assignment to `count`. This can be done in many ways, but the easiest is to hit PgUp twice giving figure 3.12.

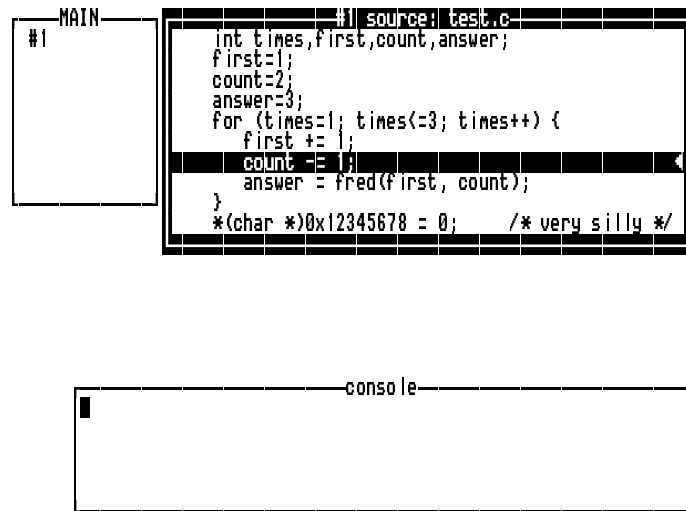


Figure 3.13

Set a breakpoint on this line by hitting ‘B’. As you can see in figure 3.13, the line is marked with a solid triangle at the right end to show you that there is a breakpoint on that line.

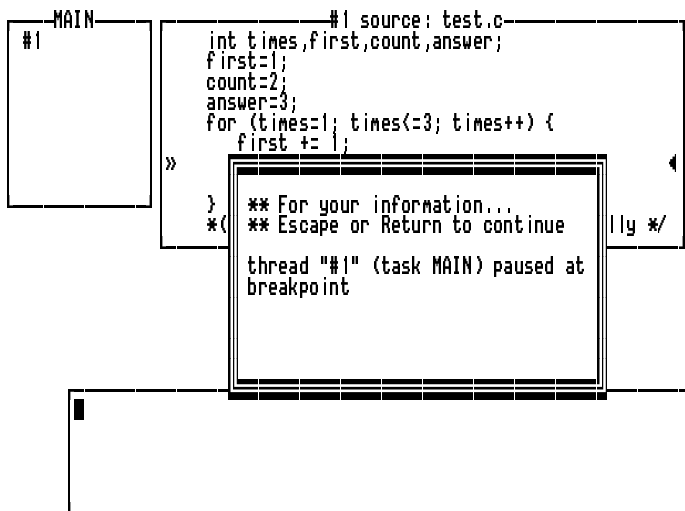


Figure 3.14

You can now let the program run freely by hitting ‘G’ (Go). As soon as the breakpoint is reached the program will be paused and you will get an *alert* window on the screen to show you that you have stopped at a breakpoint. Figure 3.14 shows you the alert window.

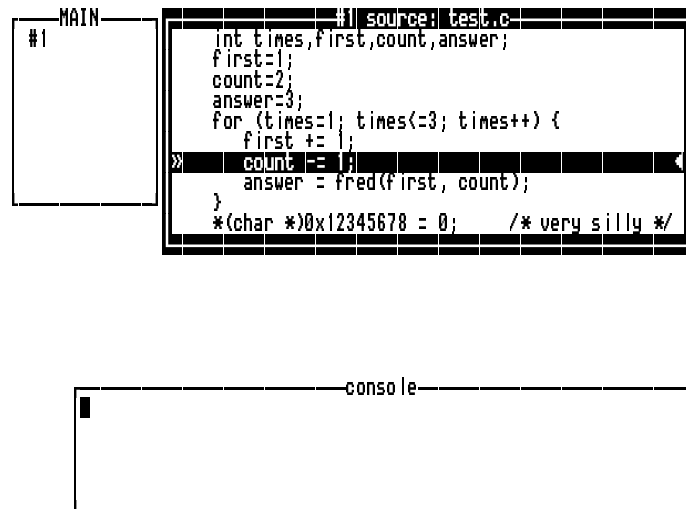


Figure 3.15

To get rid of the alert you can do several things. The window itself gives you two options: Escape (Esc) or Return (Enter). Esc is used to “give up” the current operation and Enter usually indicates successful completion of the operation. In addition, you can also hit Del, which can nearly always be used to get rid of windows. In this case, the three keys have the same effect: the alert disappears. This will leave you with a screen like figure 3.15 with the cursor on the line with a breakpoint.

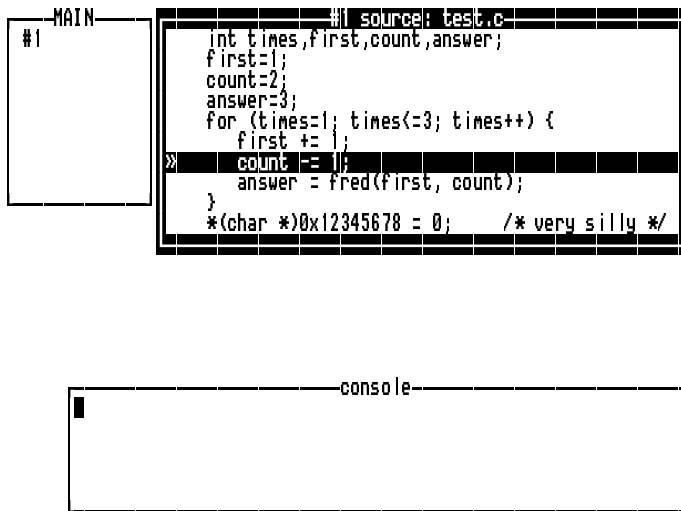


Figure 3.16

As we do not wish to stop at this point again you can remove the breakpoint by pressing 'B' once more. This will remove the breakpoint and also the indicator, giving you figure 3.16.

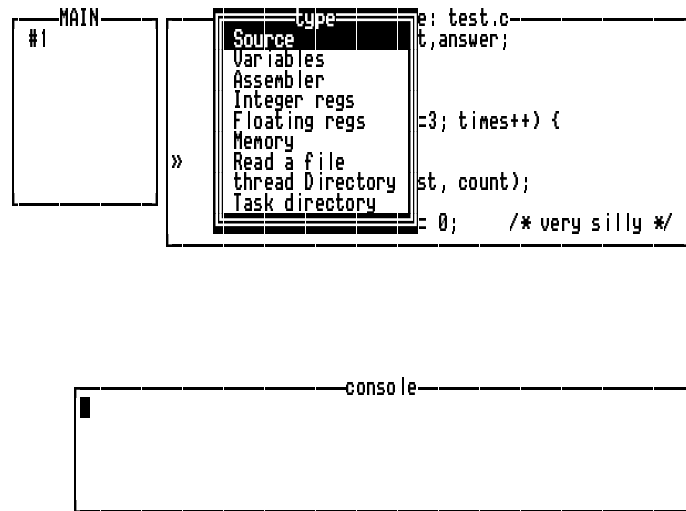


Figure 3.17

Let us now look at some variables. Hit `Ins` to bring up the screen in figure 3.17.

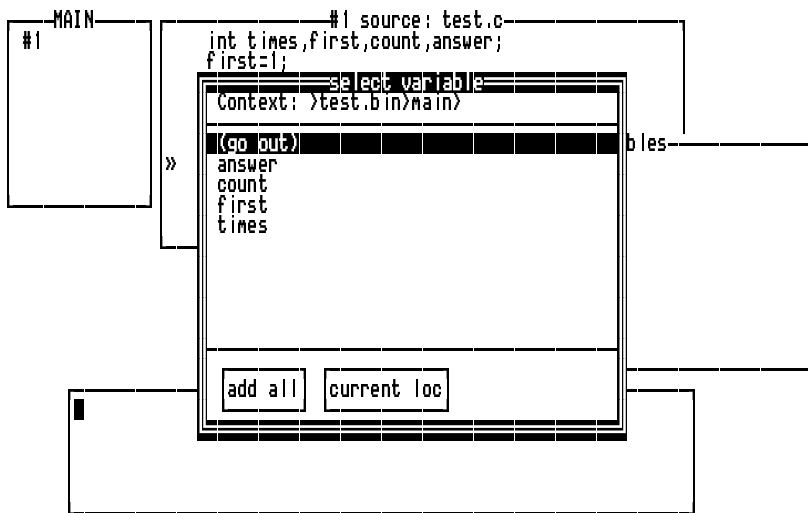


Figure 3.18

This is a *thread window menu*. Select the “Variables” option by hitting ‘V’. You should now have a list of the variables in the current function as shown in figure 3.18.

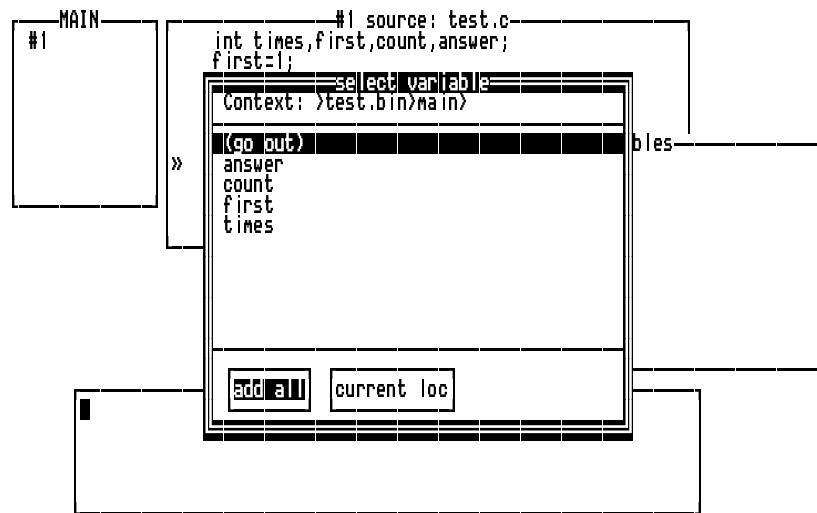


Figure 3.19

This is called a *variable selection menu*. Hit Tab and you will get figure 3.19.

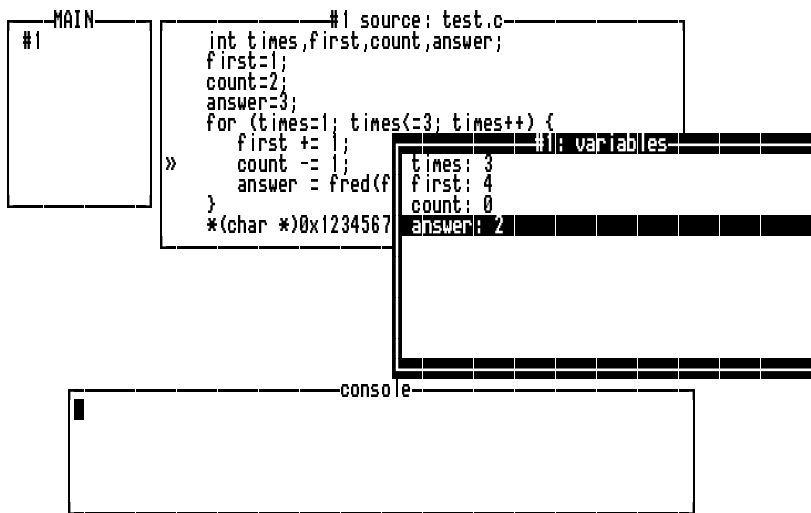


Figure 3.20

As the cursor is now on the `add all` option, hit Enter and your variables window will display all of the variables, as you can see in figure 3.20.

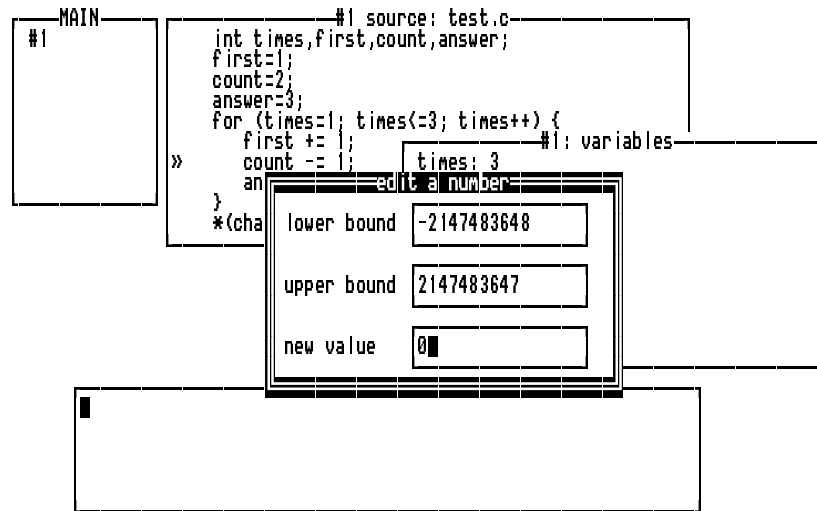


Figure 3.21

Not only can you observe the values in the variables but you can also change them. Let's change `count`. Move the cursor onto `count` by pressing `↑` once and then hit `E`. This will give you an *edit a number* window like figure 3.21.

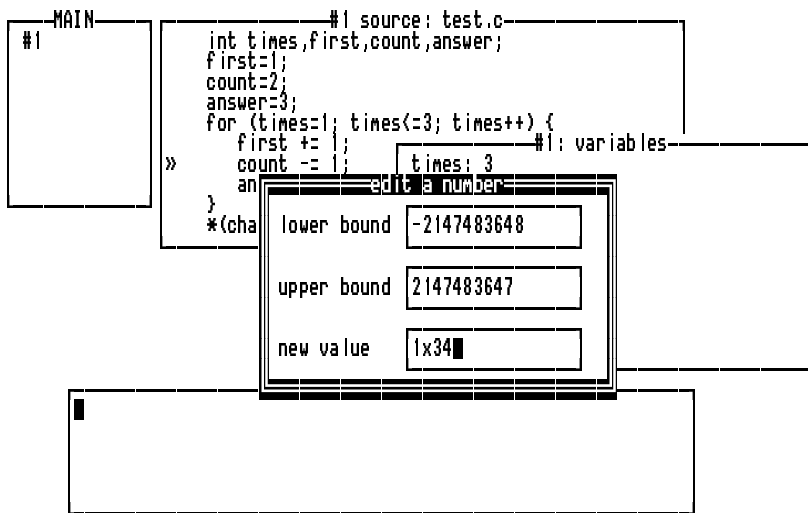


Figure 3.22

The current value of `count` is displayed in the bottom box with a cursor. Change the value by typing: Backspace, '1', 'x', '3', and '4'. This should give figure 3.22. Very confusingly, most keyboards show Backspace as . Do not confuse this with the left arrow cursor control key. Backspace is often above the Enter key.

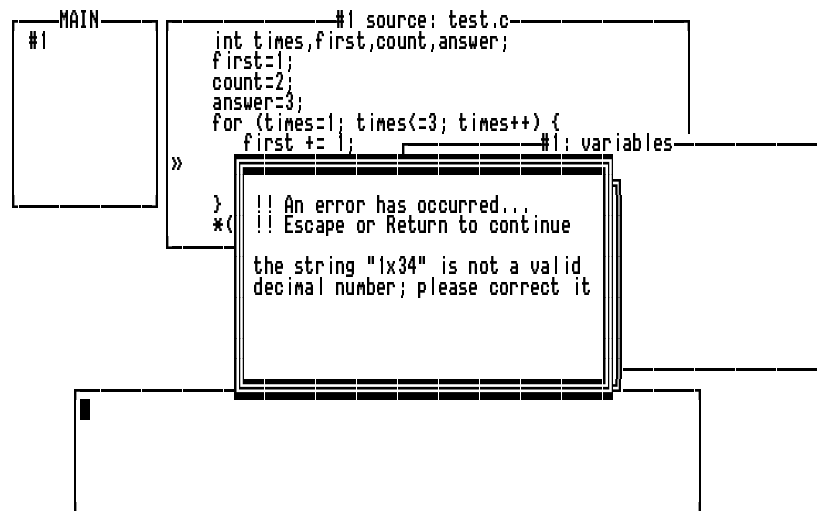


Figure 3.23

You can now try to put this value into `count` by hitting Enter. Not surprisingly, as the value is not a proper number, you will get an error as shown in figure 3.23.

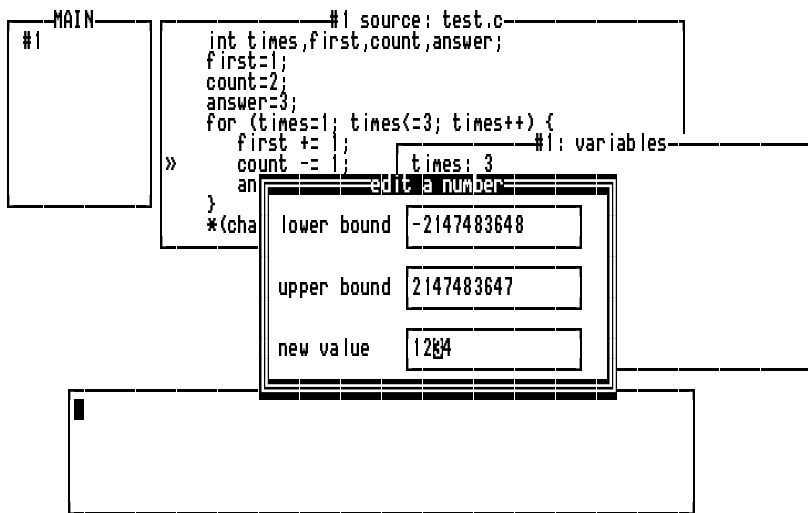


Figure 3.24

Get rid of the error alert in the usual way by hitting Del. Now put the cursor to the right of the error ('x') bar by hitting '←', '←', and then delete the offending character with Backspace. Type in the correct digit by hitting '2' to give you figure 3.24.

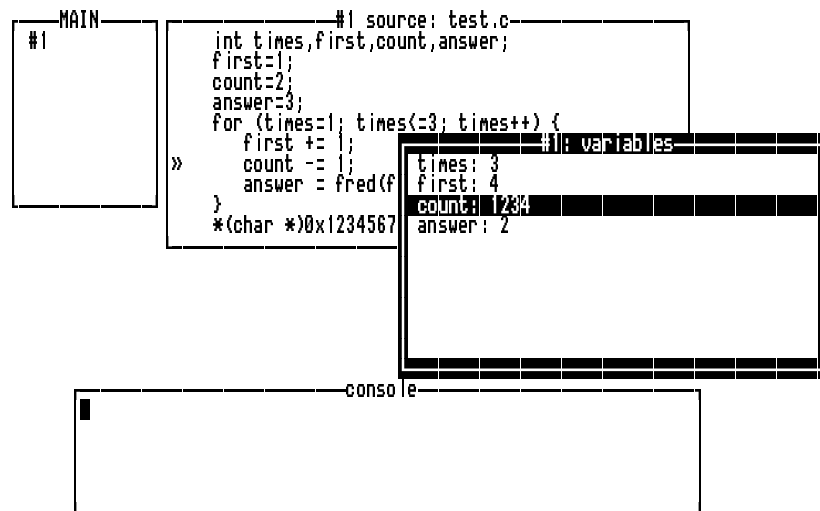


Figure 3.25

Enter will now accept the number and update the value of count as shown in figure 3.25.

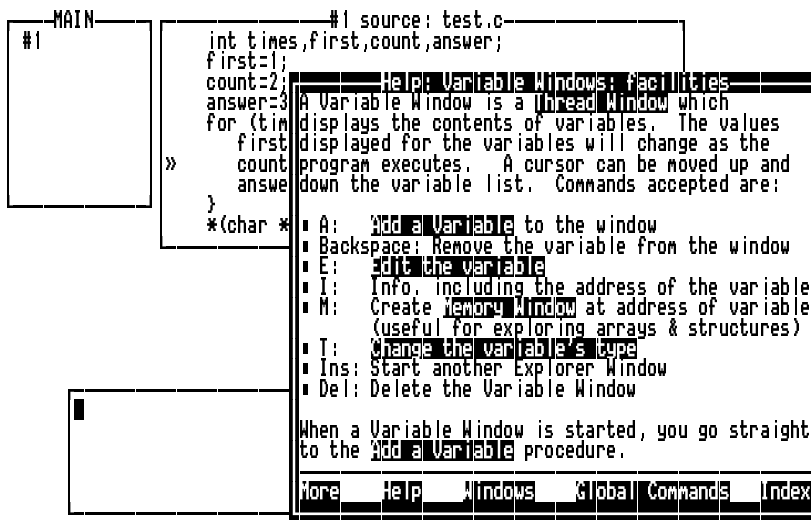


Figure 3.26

If you get stuck while debugging a program you can usually get help by typing F1. Try it now and you should get figure 3.26.

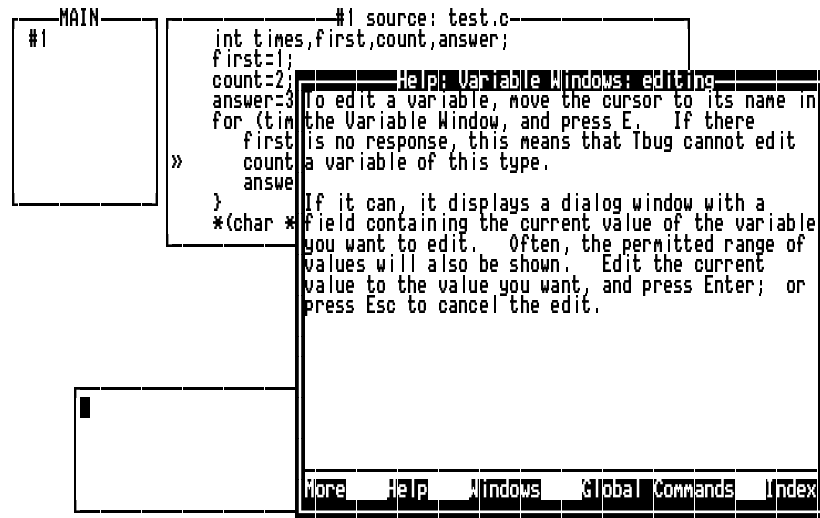


Figure 3.27

You will see that the help information has several highlighted words and phrases. These are *references* to other help pages. You can look up a reference in two ways: either by replacing the help window with the reference or creating an additional help window. Move the cursor onto the “Edit the variable” reference by hitting ‘↓’ twice. If you wanted to replace the current help page you could hit Enter, but instead hit Ins. You will get a new page (on top of the old one) with the reference text as shown in figure 3.27.

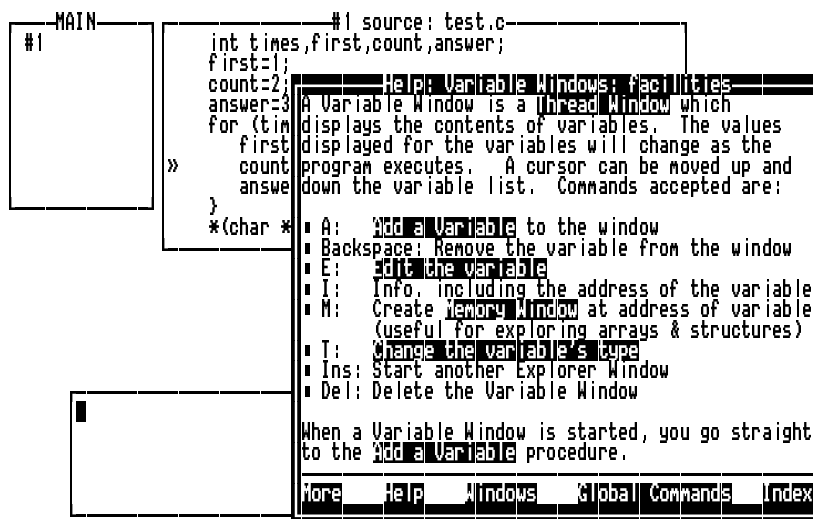


Figure 3.28

After reading the help information you can get rid of the page in the usual way by typing Del. This will bring back into view the previous page, which was underneath, giving figure 3.28.

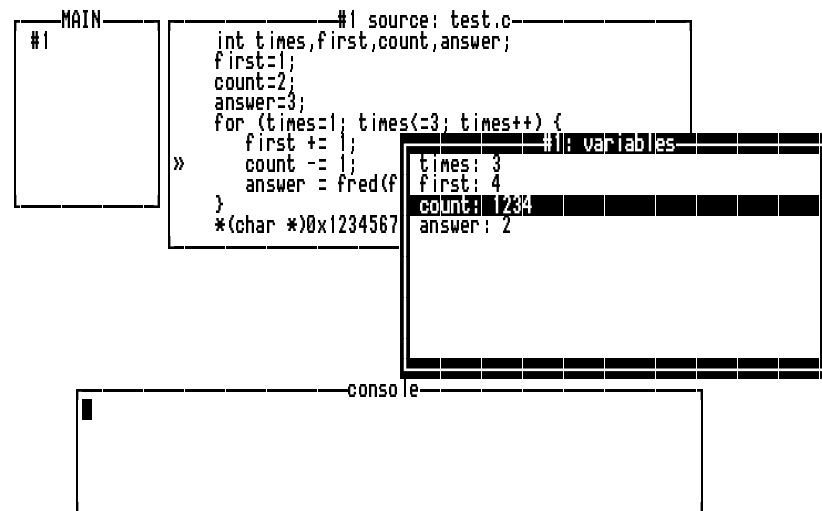


Figure 3.29

Hit Del again and the help page will disappear, leaving the screen back at figure 3.29.

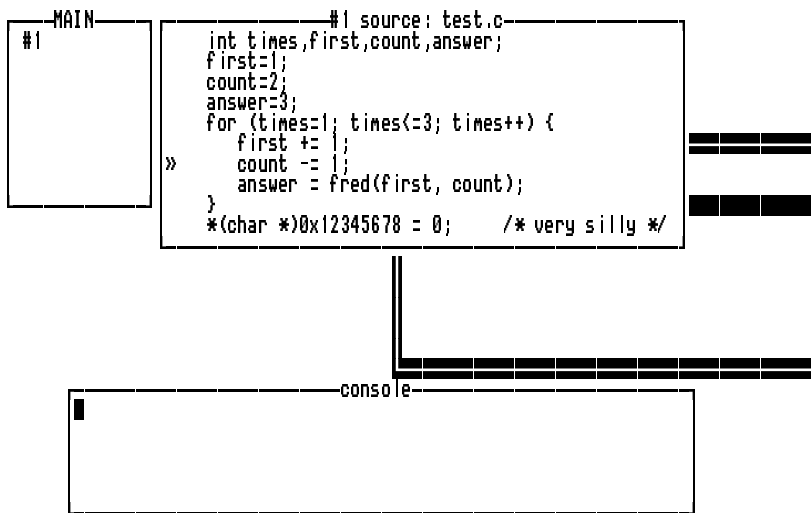


Figure 3.30

You do not have to leave windows in the position in which Tbug creates them. F7 and F8 will move the selected window to the “front” and “back” of the other windows. Hit F8 to push the variables window behind the others, as shown in figure 3.30.

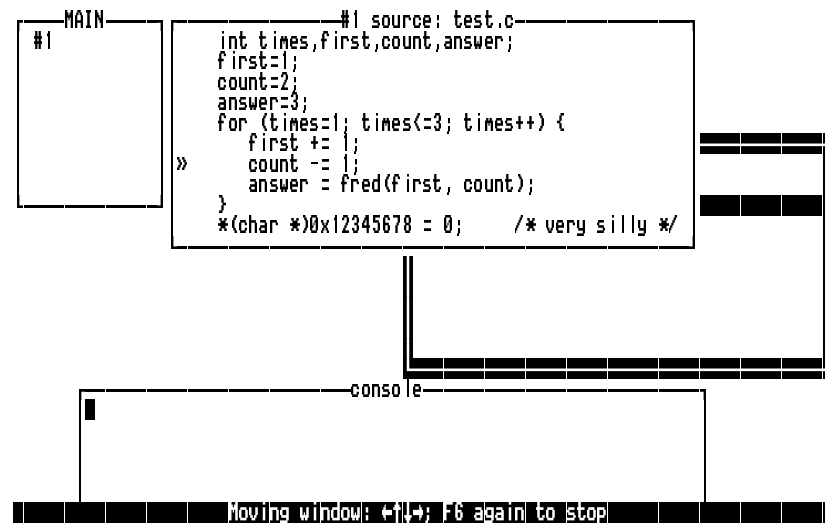


Figure 3.31

This is not a convenient position, but now hit F6. This puts you in *moving window mode* as figure 3.31 shows.

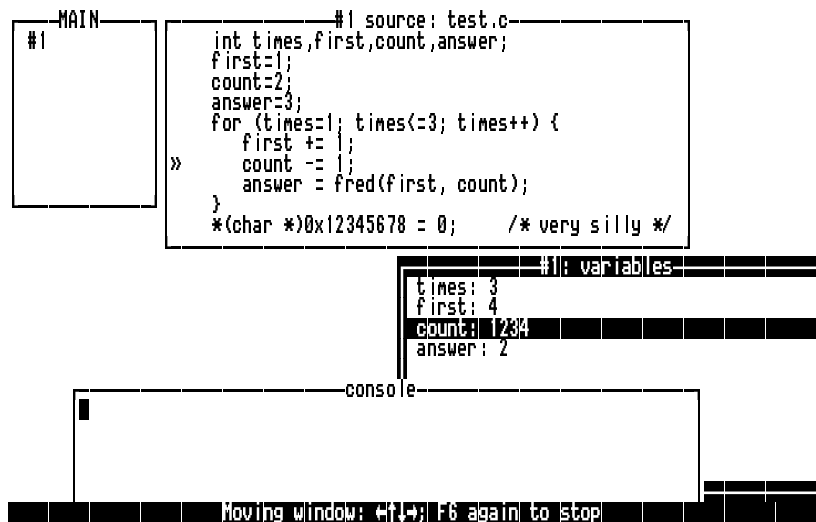


Figure 3.32

If you now hit '↓' six times you will get figure 3.32 where the window is visible properly again.

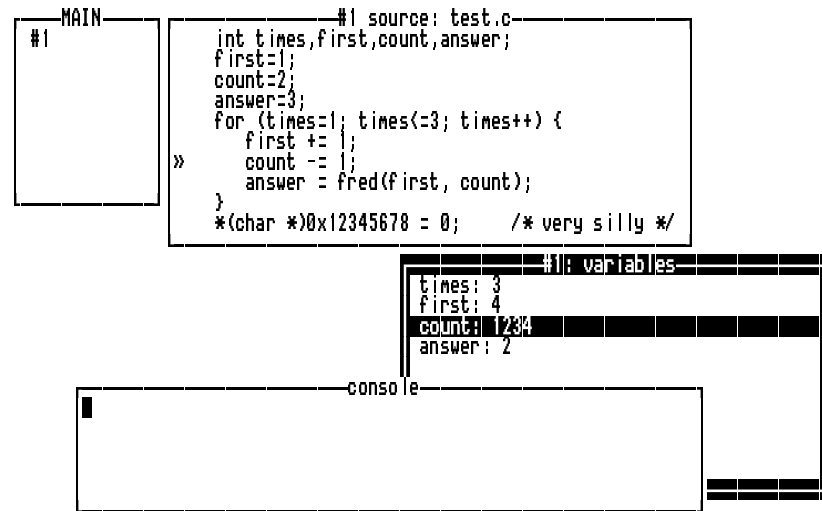


Figure 3.33

Now that the windows are in useful positions, get out of moving window mode by pressing F6 once more. The bar at the bottom of the screen will disappear leaving figure 3.33.

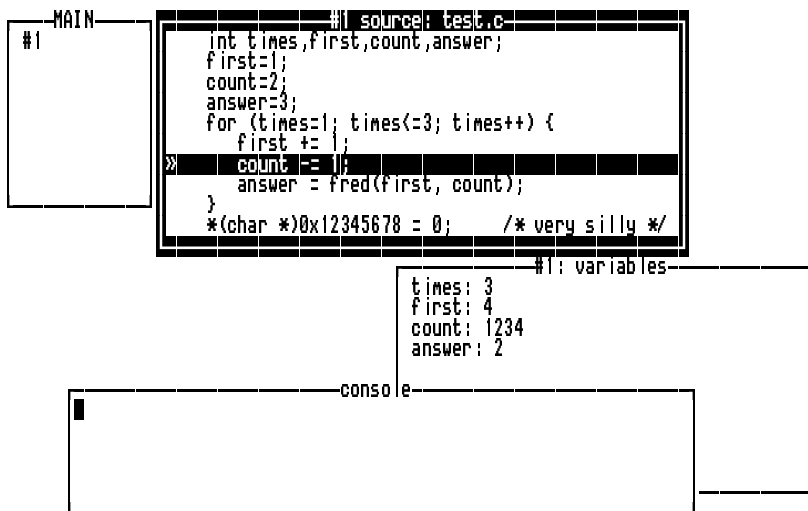


Figure 3.34

We can continue to execute the program by selecting the source window with F5. This will highlight the window as in figure 3.34.

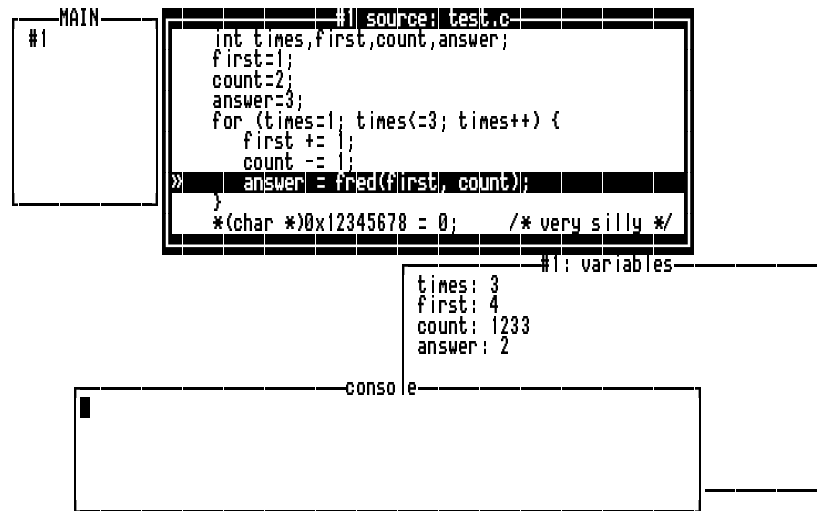


Figure 3.35

Hit the space bar and watch the value of `count` change in the variables window to give figure 3.35.

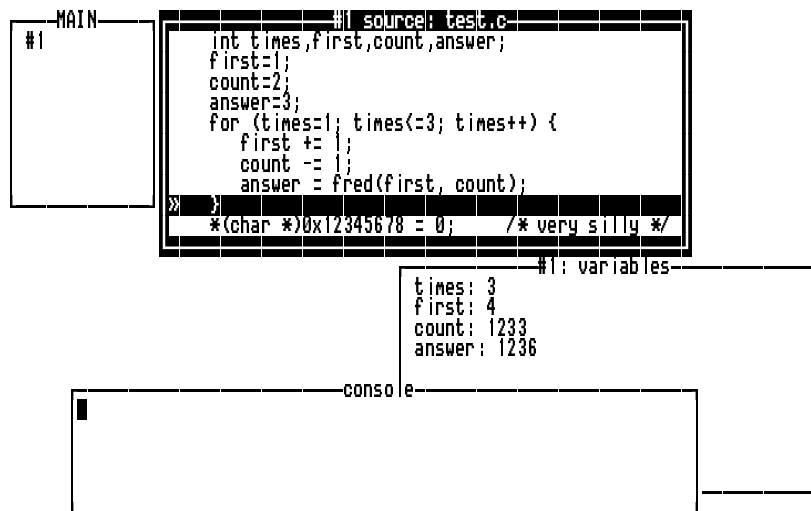


Figure 3.36

Hit space again and get figure 3.36.

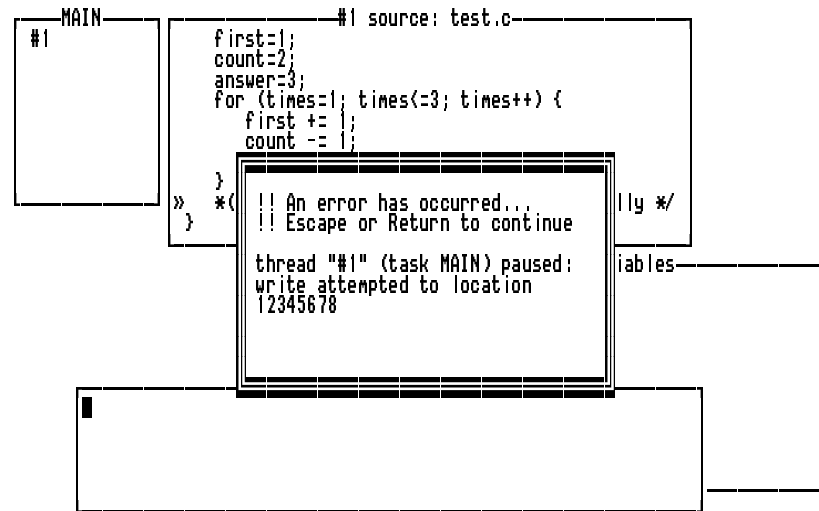


Figure 3.37

Finally, let the program run freely by hitting ‘G’. As the program has a deliberate error you will get figure 3.37 in which you can see an error alert.

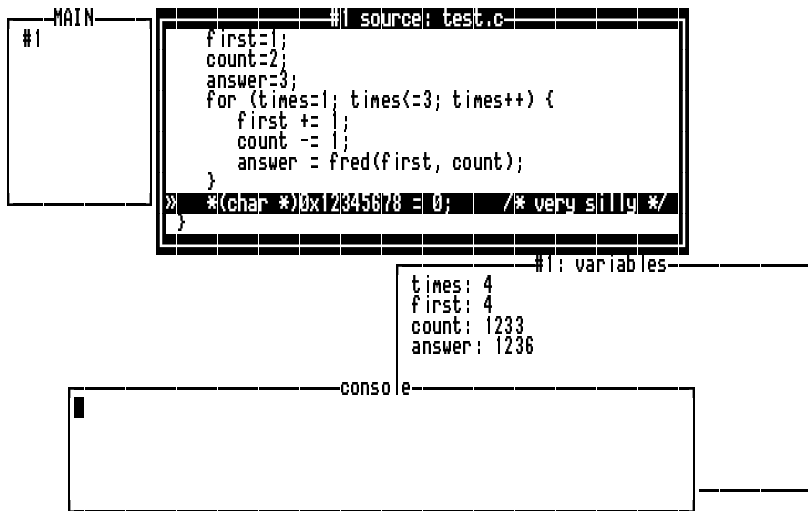


Figure 3.38

You can remove the alert by hitting Del to show the final state of the program as in figure 3.38.

To finish off the session, type Alt-X to stop Tbug and return to the MS-DOS command prompt.

Chapter 4

Compiling and Linking

If Tbug is to make all its facilities available to the user, it needs two special sets of information, known as *debug tables*, which it gets from the executable program file. These debug tables are:

1. The *line-number tables*. These contain the necessary information to marry the executable code of the program to the lines of the source file. If they are not present, Tbug will be unable to display the source of program, and the user will have to resort to assembler-level debugging, as described in chapter 8.

The line-number tables are also used by the **decode** utility.

2. The *variable tables*. These contain the names, locations and types of the variables used in the program. If they are absent, the Tbug user will not be able to identify internal variables by their names.

It is worth noting that the inclusion of the debug tables does not make the program execute more slowly outside Tbug, although the executable file will be larger.

4.1 Compilation

Both kinds of debug table are initially created by the compiler.

Currently, all 3L compilers generate line-number tables and include them in the object files they output. In the future, however, this may not be the case, and for this reason, the version 2.1 compilers recognise a `/Zd` switch, which may later cause the line-number tables to be generated. Users of these compilers may find it helpful to get into the habit of using this switch.

By default, the 3L compilers do not generate variable tables. The version 2.1 compilers, however, have a `/Zi` switch which causes this information to be produced. The version 2.0 compilers do not have this switch.

As an example, the following command would compile a C program and include both the line-number and the variable tables:

```
C>t4c test /Zi/Zd
```

Further information about the use of the 3L Parallel compilers may be found in the appropriate *User Guide*[3][4][5].

4.2 Linking

In order for Tbug to be able to make use of the debug tables which the compiler has generated, they have to be included in the executable file. Only version 1.0 (and later) of the 3L Linker is able to do this; this is distributed with Tbug and with version 2.1 of Parallel Fortran. The linkers distributed with the other compilers cannot generate debug tables.

By default, the 3L Linker does not in fact pass the debug tables on to the executable file; this is to keep the size of executable files to a

minimum in the ordinary case. To make it pass on the debug tables, you must use the `/G` switch. For example:

```
C>linkt test \tc2v1\crt1t4 \tc2v1\t4harn /g
```

If you are building a library of compiled modules, and you wish to be able to debug the members of this library when they have been incorporated in an executable program, you must also use the `/G` switch when you use the linker to make the library. For example:

```
C>linkt subs1 subs2 graphix mathpack, mylib /l /g
C>linkt test mylib.lib \tc2v1\crt1t4 \tc2v1\t4harn/g
```

It is worth noting that the run-time libraries distributed with the compilers were not built using the `/G` switch. As a result, you cannot use Tbug to trace execution through the run-time library functions at high-level.

For further information about the 3L Linker, see the *Linker User Guide*[6], or the appropriate chapter of the *Parallel Fortran User's Guide*.

Chapter 5

Starting Tbug

The most general way to start Tbug is to give the following command:

```
C>tbug
```

This will invoke Tbug with no particular program to be debugged, and so Tbug will start by displaying a *program selection menu*, that is, a list of all the available programs and directories in the current directory. Tbug will highlight the names of directories to distinguish them from program names. You can select the program to be debugged by placing the cursor over the program name and hitting Enter. You can change to another directory by selecting its name in the same way.

If you know the name of the program you wish to debug you can start Tbug with a command of the following form:

```
tbug switches program-name parameters
```

where:

switches is one or more of the Tbug switches discussed in section 5.1 below.

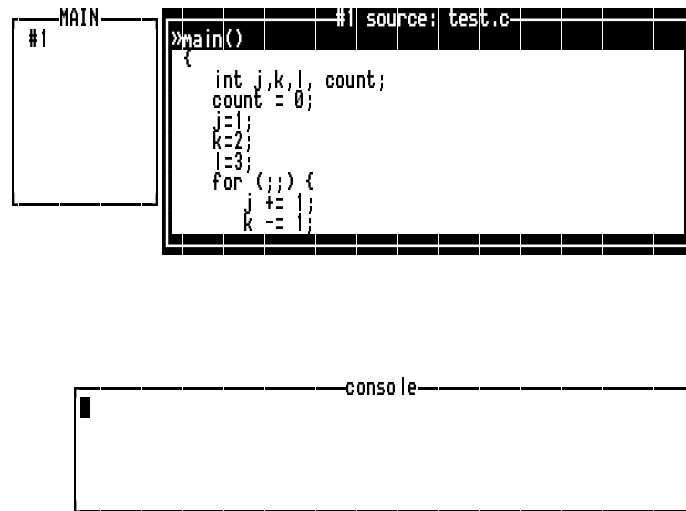


Figure 5.1: Initial Screen

program-name

is the name of the program file.

parameters

is a specification of any command-line parameters that your program needs.

For example:

```
C>tbug test.b4 data.in results.out
C>tbug /i calc.b4
```

Once started in this way, Tbug will load the program and pause at the first user instruction. You can change this behaviour by setting *preferences* as described in section 11.3. Figure 5.1 shows a typical initial screen: see section 6.2.1 for a discussion of this.

It is not possible to restart a program within Tbug; instead, you have to return to MS-DOS and invoke Tbug again, as described above.

5.1 Option Switches

The following switches may be added to the Tbug command.

- /I This switch causes Tbug to put up a window displaying its identity, including the version number. This version number should be quoted if you need to communicate with 3L about the program. Note that if Tbug is invoked with a /I switch but without any parameter, it will not display a list of programs as described above, but will instead display its identity and then stop. See also section 6.5.1.
- /P This switch causes Tbug to display the *preferences window*, enabling you to adjust Tbug's behaviour to suit your preferences. A full discussion of this facility may be found in section 11.3.

5.2 Configured Applications

This description has so far dealt only with non-configured, single-task applications. If you are debugging a configured, multi-task application, you must use a slightly different procedure to start Tbug; this is described in section 9.2.1.

Chapter 6

The Window System

The window system is the part of Tbug which controls its screen display. It supports all standard IBM text video modes and lets you interact with a potentially large number of different objects in a simple and controlled way.

If you have worked through the example session in chapter 3 you will be fairly familiar with the window system by now. This chapter is intended primarily for reference.

The window system supports three types of window, each of which behaves slightly differently. They are:

- Ordinary windows
- Help windows
- Special windows, or dialogs

These three types of window will be discussed shortly.

6.1 Basic Window Operations

6.1.1 Selecting Windows

At any stage during a debugging session there will always be one window which is the current focus of attention. This is known as the *selected window* and is highlighted in some way to distinguish it from the others; often this is done by increasing the intensity of the window's border.

The function keys F5 and Alt-F5 change the selected window by cycling round the windows on the screen. Each time you press one of these keys, another window will be selected. When you press F5, the next oldest window is selected; when you press Alt-F5, the next newest window is selected. In each case, if there is no such window, it cycles back to the beginning again. Shift-F5 may be used as an alternative to Alt-F5.

Your keystrokes are normally processed by the selected window. If the selected window does not define a meaning for a keystroke it receives it will pass it on to be interpreted as a command which is independent of the current window. Such commands are known as *global* commands, and are discussed in section 6.5 below. A keystroke will be silently ignored unless either it is recognised by the current window or it is a global command.

6.1.2 Overlapping Windows

When one window overlaps another, part of the second window cannot be seen. The window which is displayed in full is said to be *in front* of the other. Pressing function key F7 will bring the selected window to the front, so that it will not be obscured by any other window. Similarly, pressing function key F8 will send the selected window to the back of a group of overlapping windows.

6.1.3 Moving Windows

When a window is created, Tbug will attempt to place it so that it obscures as little of the other windows as possible. At any time you can change the position of the selected window by entering *moving window mode*. This mode is entered by pressing F6; a message will appear on the bottom line of the screen to confirm that you are in moving window mode. You may now move the selected window with the arrow keys. The window system will not let you move part of a window off the screen, and so movement is limited by the screen boundaries. While in moving window mode you can perform many of the usual Tbug operations; in particular, you can select another window and adjust its position. When you have rearranged the windows to your satisfaction you can leave moving window mode by pressing F6 again.

6.2 Ordinary Windows

At any one time Tbug may be displaying one or more *ordinary windows*, each of which will give information about a particular aspect of the program being debugged. For example:

- The *console window* displays the program's output.
- The *task directory* (see section 9.2) contains a list of all the tasks in the application being debugged.
- *Thread directories* (see section 9.1) each contain a list of the threads currently active within a particular task.
- *Thread windows* display some aspect of a single thread to which they are connected. Section 6.2.4 lists the various kinds of thread window, and they are discussed in full in chapters 7 and 8.

6.2.1 The Initial Display

Tbug will automatically create some windows for you when it starts up. These initial windows are:

- The console window.
- A thread directory. In the case of a configured, multi-task application, this will be the thread directory for the task which is connected to the filter task (see chapter 9).
- If the executable file contains line-number tables (see chapter 4), a source window. If Tbug cannot find the appropriate source file (see section 7.1) this window will be empty and marked `source not available`.
- If the executable file has no line-number tables, an assembler window (see section 8.1).
- If the application is a configured, multi-task application, a task directory.

6.2.2 Creating Ordinary Windows

The general method for creating a new ordinary window is to select another ordinary window and press the Ins key. This works slightly differently for the different kinds.

- If Ins is pressed when the console window is selected, then a thread directory is created, or, if this is a configured, multi-task application, a task directory.
- If Ins is pressed when the task directory is selected, a thread directory is created. This thread directory corresponds to the task under the task directory's cursor bar at the time.
- If Ins is pressed when a thread directory is selected, a thread window menu is displayed. This enables you to choose a variety

of thread window to create. The new thread window will be connected to the thread under the thread directory's cursor at the time.

- A thread window menu is also displayed if Ins is pressed when a thread window is selected. In this case, the new thread window will be connected to the same thread as its 'parent' window.

6.2.3 Deleting Ordinary Windows

You can delete any ordinary window by selecting it and then pressing the Del key. There is one exception: you cannot delete the console window.

6.2.4 Thread Windows

As we saw above, a thread window displays some aspect of the one thread to which it is connected. Thread windows are the most important tools for examining and controlling the program's execution.

The description of ordinary windows above means that you can raise a thread window in one of two ways:

- select another thread window which is connected to the thread you are interested in, and press the Ins key;
- select a thread directory, move the cursor to the thread you want and press the Ins key.

This will raise a *thread window menu*, which contains a list of all the varieties of thread window. Figure 6.1 shows an example. You can choose the variety you want in one of two ways:

- Move the cursor to the variety you want, and press Enter.

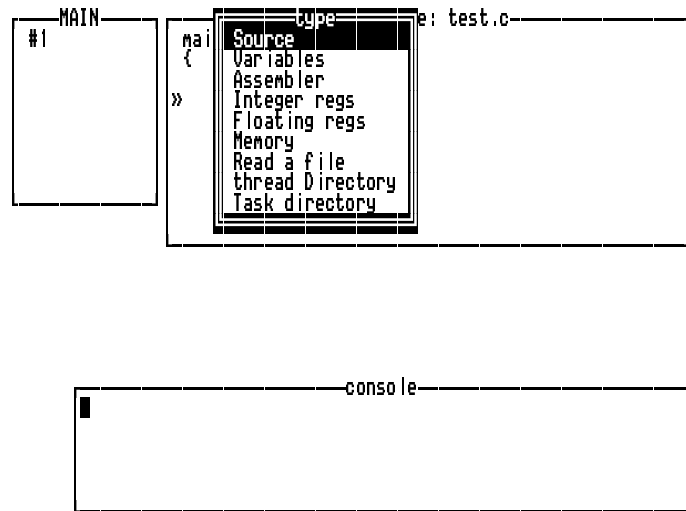


Figure 6.1: Thread Window Menu

- One letter of the variety name displayed in the menu will be a capital letter; press this key. For example, to create a variables window, press ‘V’.

The following varieties of thread window are available. They are all discussed in detail in chapters 7 and 8.

Source windows (see section 7.2) are used to display and control the program in its source form.

Variables windows (see section 7.3) are used to examine the variables which are accessible to the thread, using their source-code names.

Assembler windows (see section 8.1) are used to display and control the program in assembler mnemonics.

Integer register windows (see section 8.2.1) display the contents of the thread’s integer registers.

Floating-point register windows (see section 8.2.2) display the contents of the thread's floating-point registers.

Memory windows (see section 8.3) display blocks of memory in various formats.

File windows (see section 7.4) are used to inspect the contents of files.

Any thread window can be deleted by pressing the Del key.

6.3 Help Windows

Tbug provides you with extensive on-line help information which you can use to supplement the User Guide.

You can display a help page by selecting an ordinary window and pressing F1. The page which is displayed will contain information that is relevant to the ordinary window you selected. Only ordinary windows currently respond to F1; help windows and special windows do not.

Help pages generally contain several *references* to other help pages. These references take the form of highlighted words or phrases, one of which will be the current focus of attention and will be specially marked in some way, such as by a change in the background intensity. You can change this focus of attention to other references by means of the arrow keys, Tab, or Shift-Tab.

When you have selected a reference and wish to follow it, you can do either of the following:

- Press Enter. This will cause the current Help page to be overwritten with the page corresponding to the reference. You can use this when you do not wish to return to the original Help page.

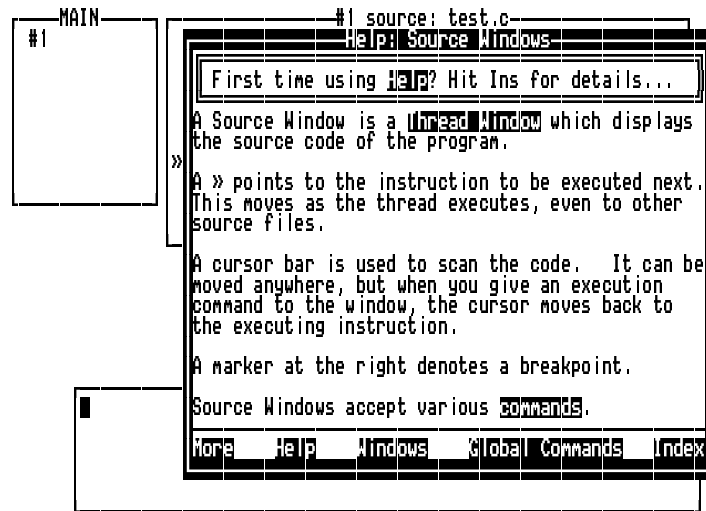


Figure 6.2: A Help Window

- Press Ins. This will create a new help window containing the page corresponding to the reference, without destroying the original help page.

When you have finished reading a Help page you can remove it by pressing Del. Figure 6.2 shows a typical help page.

6.4 Special Windows

Special windows, sometimes called *dialogs*, are distinguished visually from ordinary windows, for example by having double borders or a different colour. They require some kind of user response, and must be dealt with at once. For this reason, a special window is *always* selected, and cannot be deselected; when a special window is being displayed, F5 and Alt-F5 are inoperative.

There are many different types of special window. The methods for using them are discussed in various parts of the User Guide, as they

arise. Some general rules may be pointed out:

- When you have made your response, you should press the Enter key. This will remove the special window and make your response effective.
- For most special windows, the Esc or Del keys will cancel the operation the special window is trying to perform.
- Sometimes, a special window presents you with a number of alternatives to choose. You can do this by using the arrow keys to move the cursor, and then pressing Enter.
- Sometimes a special window is trying to perform a number of operations. Usually, these will be divided by rules within the window. You can move from one operation to another by using the Tab key, to move to the next operation, or Shift-Tab, to move to the previous operation.

Two varieties of special window are worth examining at this point.

6.4.1 Alerts

An alert window is displayed by Tbug when something important happens. The most usual thing is an error of some kind. Alerts also appear when, for example, a breakpoint happens, or as the result of a 'V' command (see section 6.5.1 below).

An alert window is displayed with some particularly noticeable characteristic, such as very wide borders or being coloured red. You have to clear it, by pressing Esc, Enter or Del, before you can continue.

Figure 6.3 shows an alert window produced as a result of giving incorrect input to a special window which was asking for a decimal number: see section 6.4.2 below.

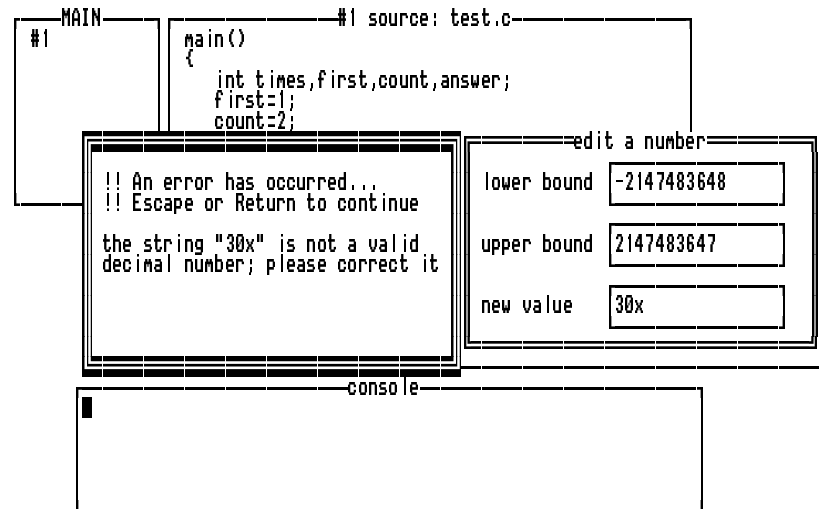


Figure 6.3: An Alert Window

6.4.2 Edit Boxes

Some special windows contain areas called *edit boxes* in which you can edit values. Inside such edit boxes the following keys have special significance:

Esc	If the edit box is empty then abandon the editing operation; otherwise empty the edit box.
Del	Delete the character under the cursor.
Backspace	Delete the character to the left of the cursor.
←	Move the cursor one character position to the left.
→	Move the cursor one character position to the right.
Enter	Indicate that the editing operation is complete. Tbug will ignore the position of the cursor when it interprets a value. Mistakes, such as incorrect characters

in numerical values, will bring up an *alert* window describing the error. When the alert window is dismissed by Del you may correct the error or abandon the edit by hitting Esc, usually twice.

6.5 Global Commands

As was mentioned above, if you press a key, it is passed first to the currently selected window, and if this does not understand it, it is assumed to be a *global command*. This sequence is, in fact, only followed when an ordinary window or help window is selected. When a special window is selected, global commands are not recognised; if you want to use one, it may be necessary to press Esc one or more times, until any special windows have been deleted.

6.5.1 Display Version Command

Pressing 'V' will create an alert window displaying the identity of the version of Tbug you are using. The version number shown in this window should be quoted in any correspondence about Tbug. The window may be removed by pressing Del. Figure 6.4 shows an alert window giving the version number 1.0.14.

6.5.2 Exit Command

The command Alt-X is used to terminate your debugging session and return you to the MS-DOS command prompt.

Sometimes you may want to leave Tbug before the initial display appears; for example, while choosing a program to debug. In this case, the Alt-X command is no use, as global commands do not work on special windows. However, if you delete the program selection menu by pressing Del or Esc, Tbug will exit.

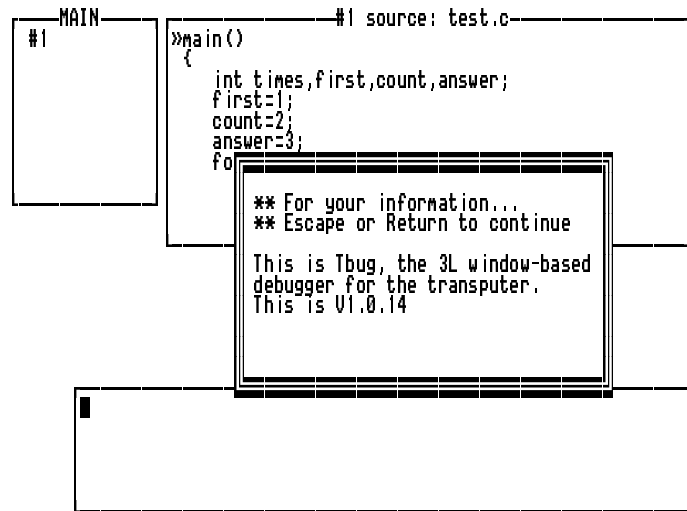


Figure 6.4: Displaying the Version Number

6.5.3 Other Global Commands

Three other global commands are discussed elsewhere. They are:

- | | |
|-------|--|
| P | Set preferences. This is discussed in section 11.3. |
| Alt-K | Invoke Master Control Panel, to change the protection settings for all the threads. This is discussed in chapter 10. |
| Alt-G | Start all threads. This is discussed in section 9.1. |

Chapter 7

Source-Level Features

7.1 Availability

For the majority of users, the facilities described in this chapter will be the most convenient to use. However, in certain circumstances, they may not be available.

Source windows will not be available if line-number tables are absent from the executable file: see chapter 4. This will be the case if:

1. The program was linked by a non-3L linker, or
2. The linker `/G` switch was not used (see section 4.2).

In addition, source windows will not work if Tbug cannot find the appropriate source file. This will be indicated by a blank source window with the title **source not available**. Tbug locates the source file for a program from information included in the program image by the compiler. If it cannot find the file in the current directory, Tbug will try to find it by following the MS-DOS search path. It will not look in the directory from which the executable file was loaded, unless it is the current directory or appears in the search path.

Information about internal variables will be unavailable if the variable tables are missing from the executable file: see chapter 4. This will be the case if:

1. The program was compiled by a version 2.0 3L compiler;
2. The program was compiled by version 2.1 of 3L Parallel C, but the `/Zi` switch was not used (see section 4.1);
3. The program was linked by a non-3L linker, or the `/G` switch was not used (see section 4.2).

7.2 Source Windows

A *source window* displays part of the source code of the program you are debugging. It is the usual way to control the execution of a program.

If possible (see section 7.1 above), Tbug's initial display will include a source window. If, at other times, you need to create a source window, you should follow the procedure described in section 6.2.4.

There are two places of special interest in a source window: the *current line*, indicated by a cursor bar, and the *execution point*, indicated by an execution marker.

In this section it is important to bear in mind that all commands given to a source window affect *only* the thread to which the window is connected.

7.2.1 Cursor Bar

The *cursor* bar defines the current line. You can move it about the source file little by little by using the following keys:

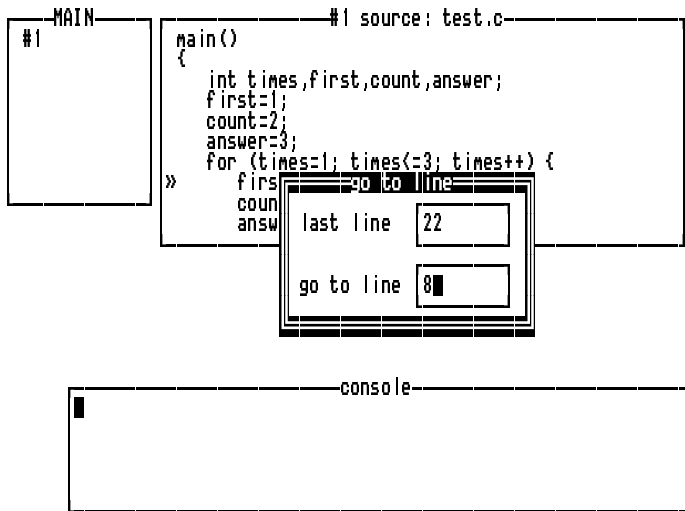


Figure 7.1: A “go to line” Window in a Source Window

- ↑ move the cursor up to the previous line.
- ↓ move the cursor down to the next line.
- PgUp move the cursor up several lines.
- PgDn move the cursor down several lines.
- Home move the cursor to the first line of the source file.
- End move the cursor to the last line of the source file.

In addition, if the source file is a long one, you might want to jump directly to a particular line. You can do this by pressing the ‘#’ key, which will bring up a *go to line* window. Type a number in the box and press Enter, and the source window will move the cursor to that line.

Figure 7.1 shows an example of a “go to line” window in a source window.

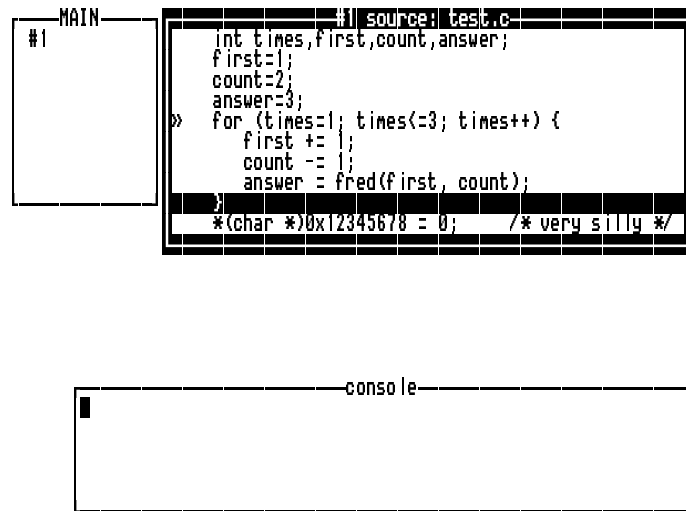


Figure 7.2: Execution Marker and Cursor Bar

7.2.2 Execution Marker

The line about to be executed by this thread is indicated by an *execution marker* in the form of a chevron (») to the left of the line. Often, this marker will be on the same source line as the cursor bar. Having moved the cursor bar you can bring it back to the execution point with ‘L’.

Figure 7.2 shows a source window where the execution marker is on a different line from the cursor bar.

7.2.3 Single-Stepping

Each time you press the Space bar the program will execute the current line and then pause. The cursor bar and execution marker will both be on the next line to be executed. This will often be the next line in the source file, but certain lines, such as the final line

of a looping statement, may cause a jump to a different part of the program.

When you use Space to step your program you will not notice calls on functions. The whole of each function will be executed as part of the current line.

Alternatively, instead of using Space, you can use Enter which has the same effect until a function call is reached. This time the function will be entered and Tbug will pause at its first instruction.

The effect of Space is termed *stepping over* the function and the effect of Enter is termed *stepping into* the function.

Execution may be suspended prematurely in either case if a breakpoint is found. Breakpoints are described in section 7.2.8.

7.2.4 Execute to Here

Instead of using Space or Enter to execute the program one line at a time, you may wish to let the program run freely until a specific line is reached. The easiest way for you to do this is to move the cursor until it is over the line at which you wish to stop and then hit ‘H’ meaning “execute up to here”. The program will start running and pause when the line is reached. The program will also pause if a breakpoint is reached on the way. Breakpoints are described in section 7.2.8. Of course, if the program never attempts to execute the selected line it will not be paused. In this case you can *break-in* to the program using ‘-’ as described in section 7.2.7.

7.2.5 Free-Running

If you give the command ‘G’, the program will start to run freely. It will pause when it reaches a breakpoint or when you “break-in” using ‘-’ as described in section 7.2.7.

7.2.6 Error Conditions

Tbug makes certain checks for error conditions occurring while the program is running. If an error is detected, the program will pause on the statement causing the error and an *alert* window will be put up describing the error. You can remove the alert window by pressing Del. Although you will now be able to examine the state of the thread, you will not normally be able to execute past the instruction which caused the error. Any other threads and tasks in your application have not, however, been affected by this and can continue executing.

The errors that Tbug can detect are:

- **address error**

The program has attempted to write to a protected memory location. By default, Tbug will prevent a program from overwriting its own code or writing to memory not in its requested allocation. This protection may be disabled if required (see chapter 10).

- **illegal instruction**

The program has attempted to execute an undefined instruction.

- **state collision**

The thread has tried to read from a channel which another thread is already reading; or the thread has tried to write to a channel which another thread is already writing to.

- **using a hard link**

The thread has attempted to perform input or output on a hard link. By default, Tbug prevents a program from doing this, but this protection can be disabled if required (see chapter 10).

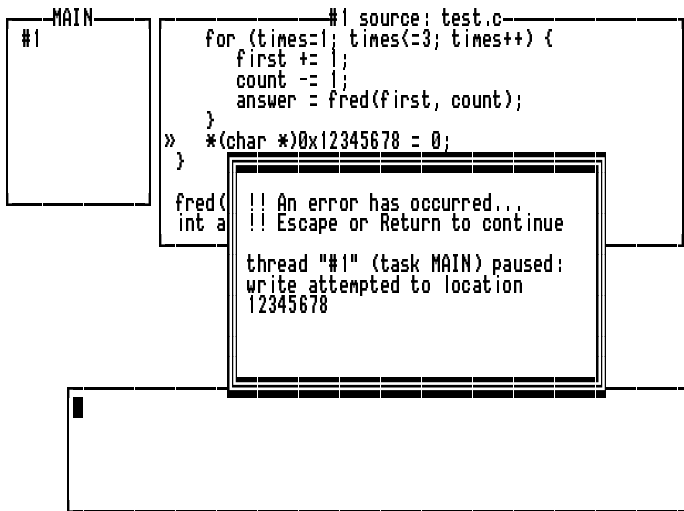


Figure 7.3: An Error Alert

- **using hard link 0**

The thread has attempted to perform input or output on hard link 0. This link is reserved for Tbug’s own use, and may not be used by the user program.

Figure 7.3 shows the result of deliberately provoking an address error.

7.2.7 Break-in

At any stage while the program is running you can make it pause wherever it is by hitting ‘-’. This applies however you started the program: by stepping over a function, executing to “here”, running it freely, or whatever. A break-in only affects the current thread, and does not affect a thread which is waiting (for example, for input or output), until it attempts to execute another instruction.

7.2.8 Breakpoints

Breakpoints provide a mechanism for making your program pause whenever it reaches particular lines. You can set breakpoints by moving the cursor onto the appropriate line and hitting ‘B’ for a *strong* breakpoint or ‘W’ for a *weak* breakpoint. You may set breakpoints on as many lines as you wish.

When you compile certain high-level language statements, declarations and comments, for example, the compiler will not generate any instructions. A consequence of this is that Tbug will not be able to place a breakpoint there. Instead, it will place the breakpoint on the next instruction it finds.

Breakpoints are associated with threads. This means that even though two threads may be sharing the same instructions, a breakpoint set in one thread will not stop the other.

Figure 7.4 shows a source window in which one weak and two strong breakpoints have been set.

7.2.8.1 Strong Breakpoints

A strong breakpoint will remain in effect until you explicitly remove it by locating the cursor over the line containing the breakpoint and hitting ‘B’ or ‘W’. You can see that a line contains a strong breakpoint by the symbol ‘!’ appearing at the right of the line.

7.2.8.2 Weak Breakpoints

Weak breakpoints remain in effect only until the next time the program pauses, at which point *all* weak breakpoints are removed. Strong breakpoints are not affected. You can explicitly remove a weak breakpoint by locating the cursor over the line containing the breakpoint and hitting ‘W’ or ‘B’. You can see that a line contains a weak breakpoint by the symbol ‘?’ appearing at the right of the line.

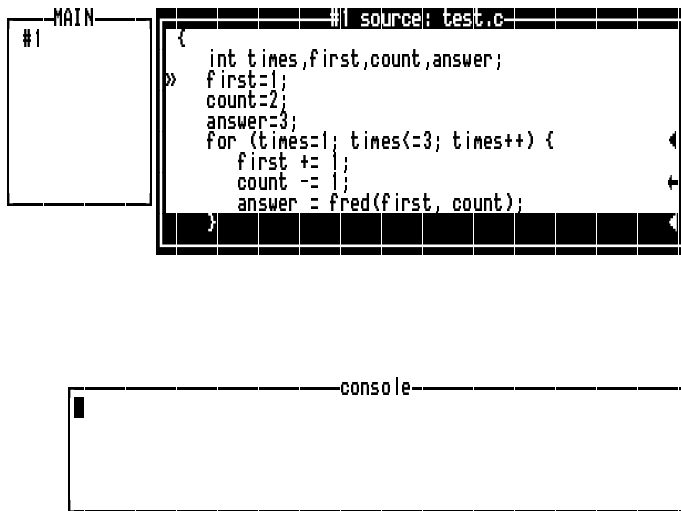


Figure 7.4: Strong and Weak Breakpoint Markers

7.3 Variables Windows

A *variables window* is a tool which enables you to observe the values of variables as the program executes, and to change them.

You can create a variables window in the way described in section 6.2.4.

7.3.1 Observing Variables

When you request a variables window from the thread window menu, you will first get a *variable selection window*, which allows you to choose which variables you want to observe. The variable selection window shows a list of all the variables defined in the current function. Figure 7.5 shows a typical list of variables.

If you wish to observe all of the variables in the list you can select the add `all` control box. You can do this by pressing `Tab`, to move the

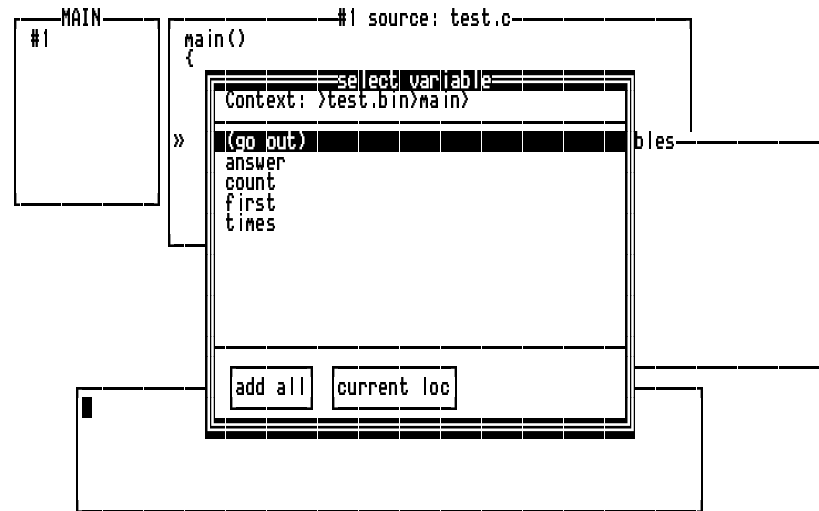


Figure 7.5: A Variable Selection Window

cursor over the ruled line, and then Enter. This will add the name of all the variables in the list to the variables window, and dismiss the variable selection window.

If you only wish to observe a few of the variables, you may select them one at a time by placing the cursor over each one in turn and hitting Enter. After each entry you will see an alert window confirming that the variable has been selected. The alert is dismissed by hitting Del. When you have selected the last one you want you can dismiss the variable selection window by hitting Del. Figure 7.6 gives an example of a variables window.

If you wish to add extra variables to an existing variables window you should select the window and hit 'A'. This will bring up the variable selection window with the list of variables so that you can choose from it as previously described.

If you use Ins a second time and select the “Variables” entry, you will not add variables to the existing variables window. Instead, you will get a second variables window.

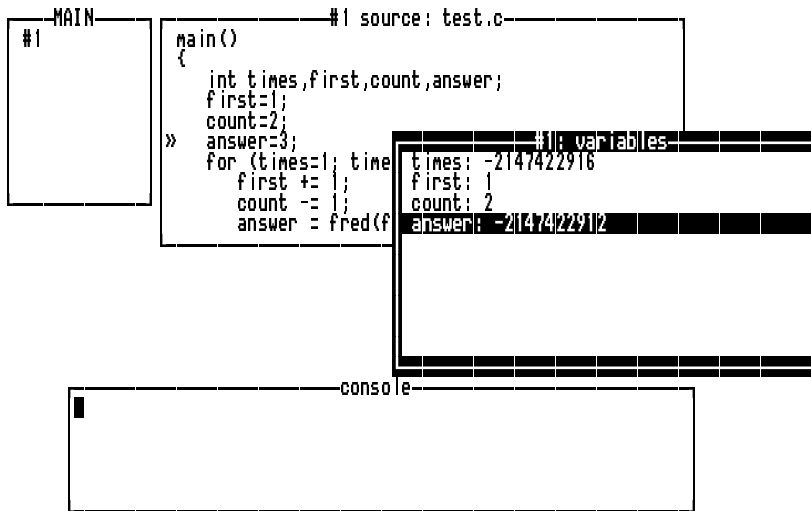


Figure 7.6: A Variables Window

You can remove a variable from a variables window by moving the cursor to that variable and hitting Backspace.

The behaviour of the variables selection window can be changed to some extent by changing the *multiple variable selection* Boolean in the preferences window. See section 11.3 for a discussion of this.

7.3.1.1 Changing the Context

The procedure described so far will enable you to observe variables which are local to the current function. Using the variable selection window, you can also change to other contexts.

An idea which will be useful at this point is the *module*. A module is the output from a single compilation, and corresponds initially to a single `.bin` file. A module may include more than one function. After compilation, modules may be grouped into libraries; and when the program is linked, a number of modules are joined together to make an executable file.

The item (`go out`) can be used to change to the context enclosing the current function. What will be visible at this point depends on the language the program was written in.

- With C, (`go out`) takes you to the context of the module. There you will see a list of the functions which form part of this module, together with external and static variables defined outside the functions.
- Similarly, with Fortran you will see a list of the subprograms which are part of this module. A main program subprogram, if present, will appear as `f_main`. `BLOCK DATA` subprograms will not appear.
- In Pascal, which supports nested definitions of procedures and functions, you should see a list of the procedures, functions and variables defined at this level. Unfortunately, version 2.0 of Parallel Pascal does not support Tbug's handling of local names, so at present with Pascal too, you move straight to the context of the module. There you will see a list of all the procedures and functions of the module, of whatever lexical level, laid out flat; local variables do not appear at all.

Variables which appear in this new context can be added to the variable window in the usual way. It is also possible to move into the context of any of the functions shown in the list, by moving the cursor to it and pressing Enter; this will enable you to inspect the variables defined within that function.

You can now access the level of a function and the level of the module. There is only one further context level available in C and Fortran, and that is the level of the entire linked program. If you select (`go out`) at the module level, you will move to this new level. Here you will see a list of all the modules which were linked together to form this program. You can move to the context of another module by moving the cursor to its name and pressing Enter.

At the entire-program level, you will also see the names of a number of external data items. They can be distinguished from the module

names, as the latter are highlighted. In C, these external data items correspond to the external variables defined at the source file level, and will have the same names. In Fortran, they will correspond to `COMMON` blocks. However, in both cases the type information is not available at this level, and they will be shown as being of unknown type.

In the case of Pascal, you might expect several levels of context, corresponding to the lexical levels of the program. Unfortunately, version 2.0 of Pascal does not support interaction with Tbug at this level, so for Pascal programs also there is only the local procedure level, the module level and the entire-program level. At the entire-program level, you will see only the module names and the names of `EXPORT` data items. Note that if you have used the `ALIAS` facility, it is the `ALIAS` name which will appear in the list of modules and external data items.

The variable selection window also has a `current loc` control box, which can be selected using Tab. This will change the context of the variable selection window to correspond to the function the thread is currently executing.

7.3.2 Changing the Values of Variables

If you wish to change the value of a variable you must first select the variables window and move the cursor bar onto the particular variable. Hitting ‘E’ will bring up a window containing three edit boxes, in which you can edit the value of the variable. Figure 7.7 shows an example of such an “edit a number” window.

Section 6.4.2 describes how you can input values using edit boxes. If you terminate the editing operation using Enter the variable will be updated, but if you abandon the edit using Esc the variable will be left with its original value.

As Tbug aims to be independent of source language conventions it is not possible to use language-dependent representations for values.

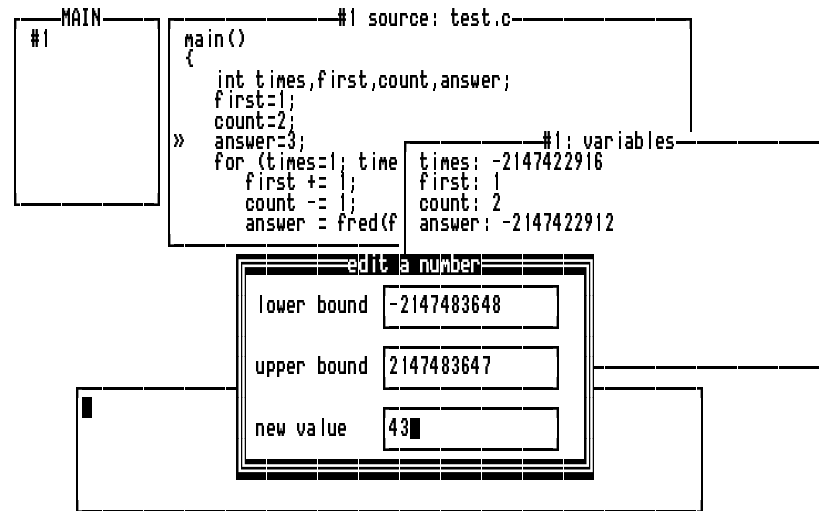


Figure 7.7: An Edit Window

In particular, C programmers should note that the denotation 012 will be interpreted as the decimal number twelve.

The present version of Tbug does not allow you to change the values of Fortran CHARACTER variables.

7.3.3 Information about Variables

You can get information about a variable by selecting it with the arrow keys and pressing ‘I’. This will create a window showing the name of the variable, its address in hexadecimal, and the context in which it was defined. You may think of this context as the equivalent of a file’s “path-name”.

7.3.4 Changing the Type of Variables

It is occasionally necessary to be able to change the apparent type of a variable. For example, the current version of the C compiler tells

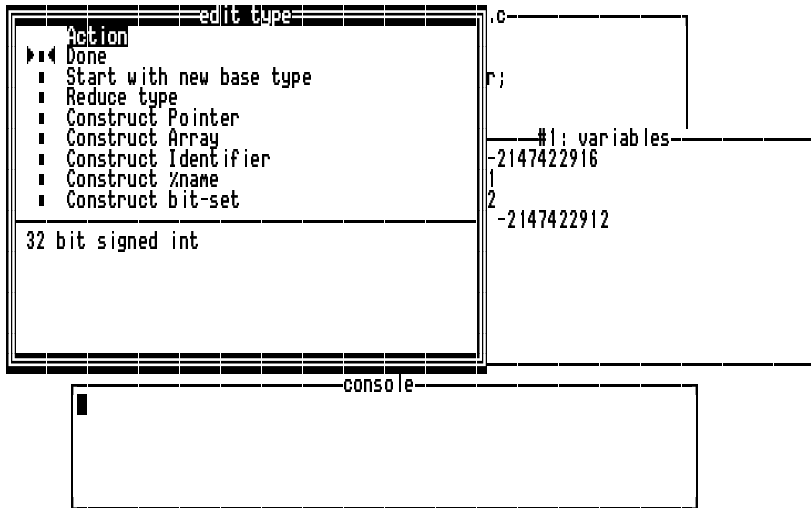


Figure 7.8: Type Editor Window

Tbug that pointer variables are simple integers (which happen to contain the address of the object being pointed to). A programmer using one of the version 2.0 compilers might also use this facility to give a type to external variables for which no type information is available.

Note that it is not at present possible to change the type of a variable to be a Fortran **CHARACTER** variable, or to be a Fortran array.

To change the type of a variable you should first select the variables window, move the cursor bar onto the variable in question, and then hit 'T'. This will bring up the *type editor* window. The lower portion of the window displays the current type of the variable while the upper portion gives a menu of options for changing the current type. Figure 7.8 shows the type editor window brought up for the variable **count**.

You can select from the menu by placing the cursor over the option and hitting Enter. The options can be applied repeatedly to build up (or reduce) complex types; the final option will always be **Done**.

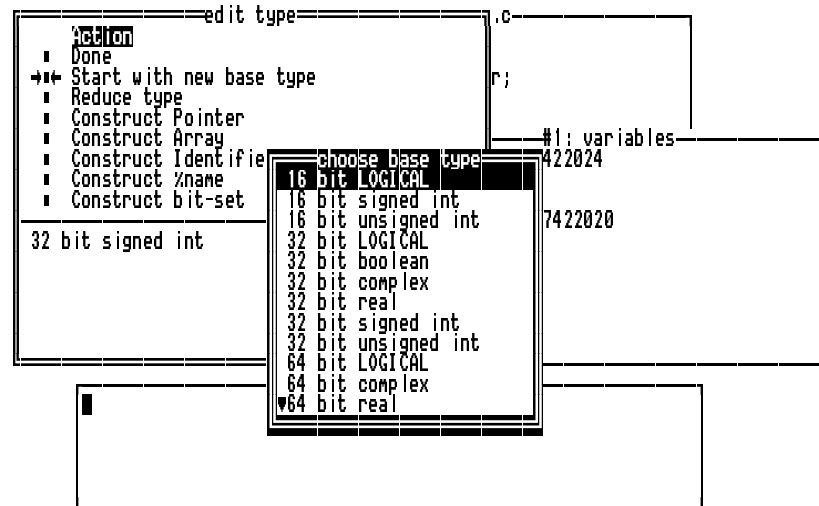


Figure 7.9: Base-Type List: Top Half

At any stage you can abandon the current operation by hitting Esc.

The options are:

- **Done**

This sets the type of the variable to the type displayed in the lower half of the window and terminates the editing operation by dismissing the window.

- **Start with new base type**

You will be presented with a new window containing a list of all the base types that Tbug understands. Figures 7.9 and 7.10 show all of the base types. You can select a particular type by moving the cursor over its description and pressing Enter. The list will be removed and the previous type of the variable will be replaced by the newly selected type.

- **Reduce type**

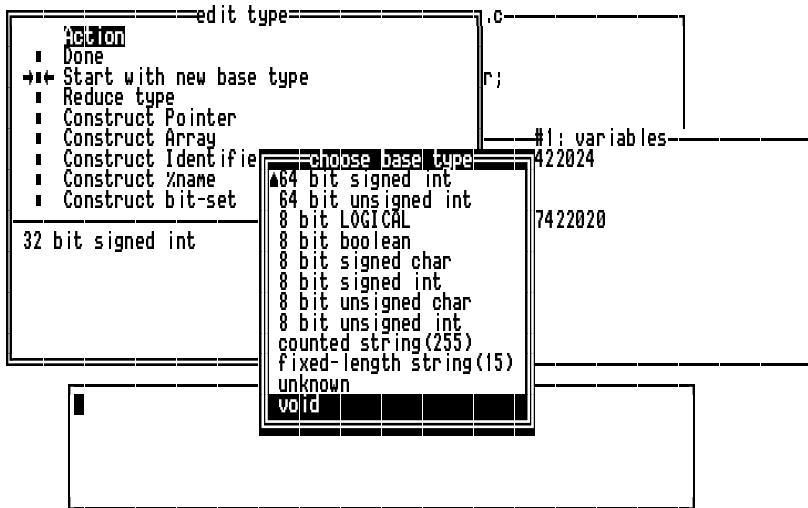


Figure 7.10: Base-Type List: Bottom Half

The leftmost component of the current type will be removed thereby simplifying the type. For example, applying reduce type to `pointer to 64 bit real` will result in `64 bit real`.

- **Construct Pointer**

The current type will be replaced by a new type which is a “pointer to” the old current type.

- **Construct Array**

You will be presented with a window in which you can edit the values for the lower and upper bounds of the array type you are creating. The current type will be replaced by a new type which is an “array[lower..upper] of” the old current type. Figure 7.11 shows an array constructor window.

For more information about the way Tbug handles arrays, see section 7.3.5 below.

- **Construct Ident**

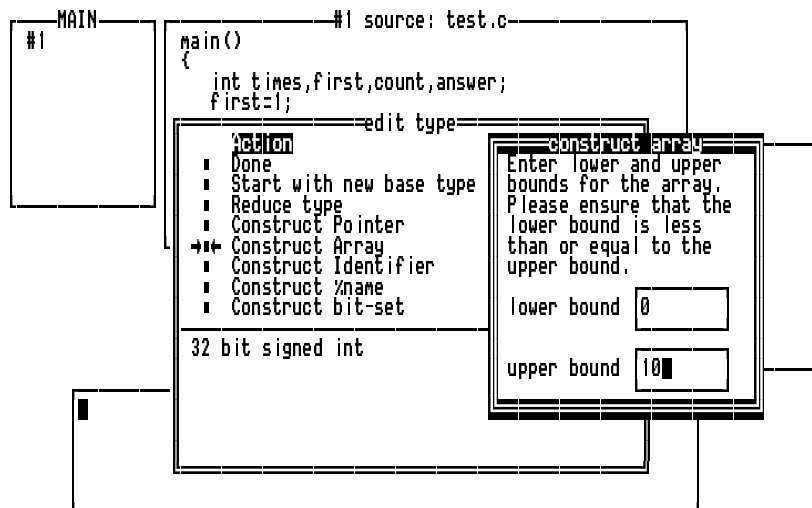


Figure 7.11: Array Constructor Window

This option lets you give an identifier to the current type.

- **Construct %name**

This option is only relevant to the Imp programming language. The current type will be replaced by the %name of the old current type.

- **Construct bit-set**

This option enables Tbug to handle Pascal SET variables. At present, every set generated by the Pascal compiler consists of 32 bytes, and is equivalent to a SET OF 0..255. The presence of item 0 in the set is indicated by bit 0 of byte 0 being set; the presence of item 255 is indicated by bit 7 of byte 32 being set. If the Pascal programmer defines a variable as a SET OF 17..23, for example, this is mapped onto items 17..23 of the underlying SET OF 0..255. Sets of CHAR map directly onto the underlying set, and sets of other types map onto a range of the underlying set corresponding to the range

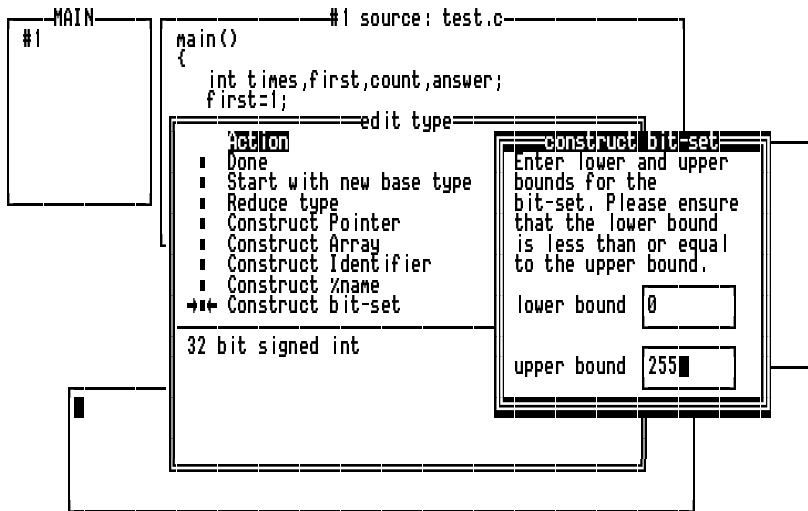


Figure 7.12: Bit-Set Constructor Window

of the ordinal values of the type. For example:

```
TYPE Country = (Scotland, England, Wales, Ireland);
VAR  CSet    : SET OF Country;
```

In this case, the variable `CSet` would occupy 32 bytes, but the 4 functional bits would map onto items 0..3 of this underlying set.

If you select this option, you will be presented with a window in which you can edit the upper and lower bounds of the ordinal values of the members of the bit-set; figure 7.12 shows such a bit-set constructor window. However, because of the way sets are implemented in Pascal, the appropriate response in every case is to leave the default bounds (0..255) unchanged.

The current version of Tbug only displays the first 5 bytes of a set, and only in hexadecimal. To look at the whole set, you must use the 'M' command (see section 7.3.6).

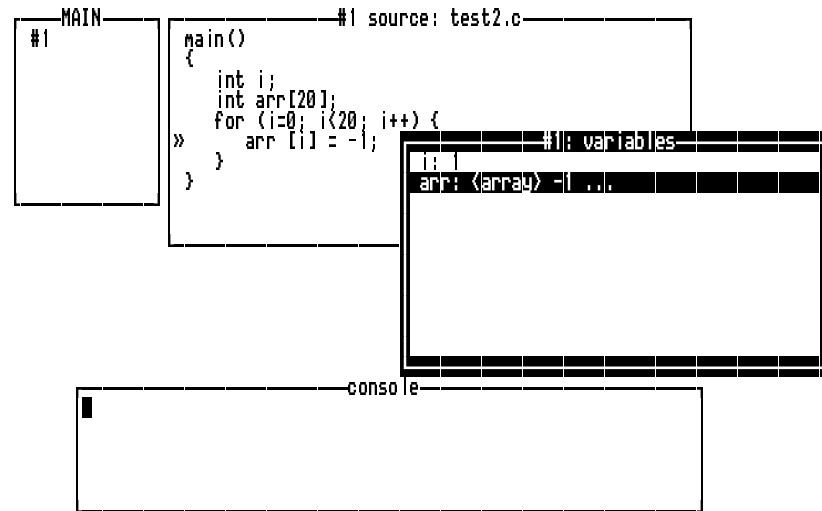


Figure 7.13: A Variable Selection Window with a C Array

7.3.5 Arrays

If you select an array to be displayed, the variable is marked in the variables window with the note `<array>`. An example is shown in figure 7.13. Fortran arrays, whether with one or more dimensions, are shown in the same way, but two dimensional C arrays are treated as arrays of arrays, and so on. A Fortran example is shown in figure 7.14.

The variables window will display only the first element of the array. To look at other elements, you must make use of an *array window*. An array window is invoked by moving the cursor in the variables window to the array you want to look at, and then pressing ‘0’.

An example of an array window, in this case examining a Fortran array, is shown in figure 7.15. You can move up and down the array using the arrow keys, PgUp, PgDn, Home and End as usual; you can also move the window from side to side, using the arrow keys. A Fortran array is laid out in the array window in the order it

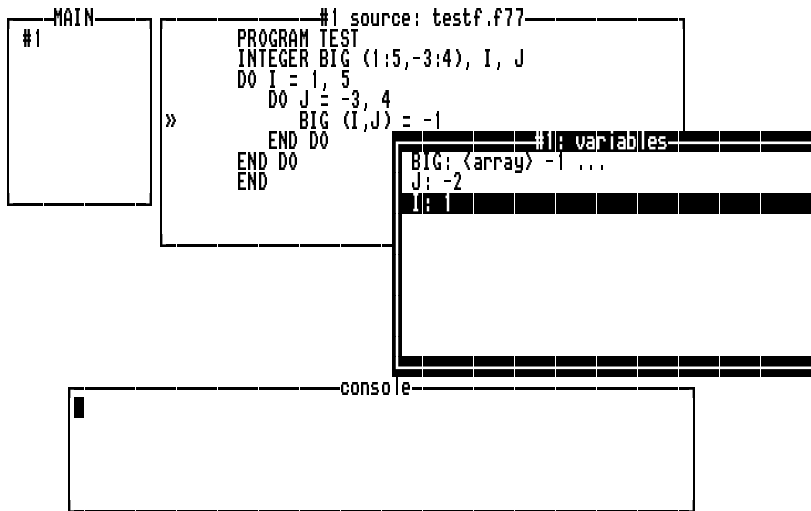


Figure 7.14: A Variable Selection Window with a Fortran Array

is held in memory; this can be seen by the subscripts displayed. Single-dimensional C arrays are displayed with subscripts running from 0; for C arrays with more than one dimension, each element is displayed as an array in its own right. Unfortunately, in the current implementation, it is not possible to open an array window on a nested array of this sort.

The value of an array element (except a nested array) can be changed in the usual way, using the ‘E’ command (see section 7.3.2 above).

You can jump around the array window by using the ‘#’ command. For this purpose, the elements in a multi-dimensional Fortran array are simply numbered in memory-resident order, from 1 up, and you will be asked which you wish to see.

As noted above, it is not possible in the current version of Tbug to change the type of a variable to be Fortran array.

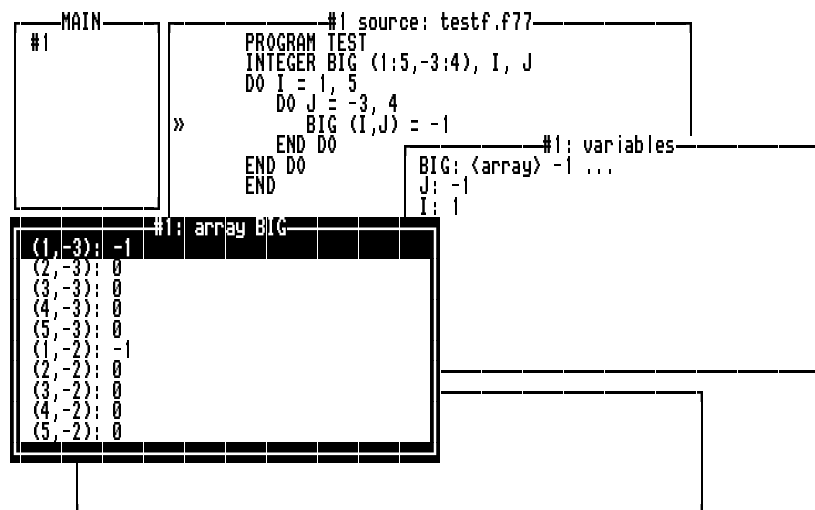


Figure 7.15: An Array Window

7.3.6 Opening a Memory Window on a Variable

If you place the cursor of a variable window on a variable and press ‘M’, you will open a memory window based on that variable. Memory windows are discussed in section 8.3. They give you the ability to view an area of memory and to display it in several different ways. This could have various uses, such as:

- The present version of Tbug does not comprehensively display the contents of C **struct** variables. Instead, they are shown as having the type **structure**, and the first 16 bytes are displayed in hexadecimal. The ‘M’ command may be used to inspect their contents a little more fully.
- As has been pointed out, the 3L compilers at version 2.0 do not support the display of internal variables. However, externally visible variables can be found, as items at the top, ‘entire-program’ context of the variable selection window. This includes Fortran **COMMON** blocks, and the ‘M’ command is a good

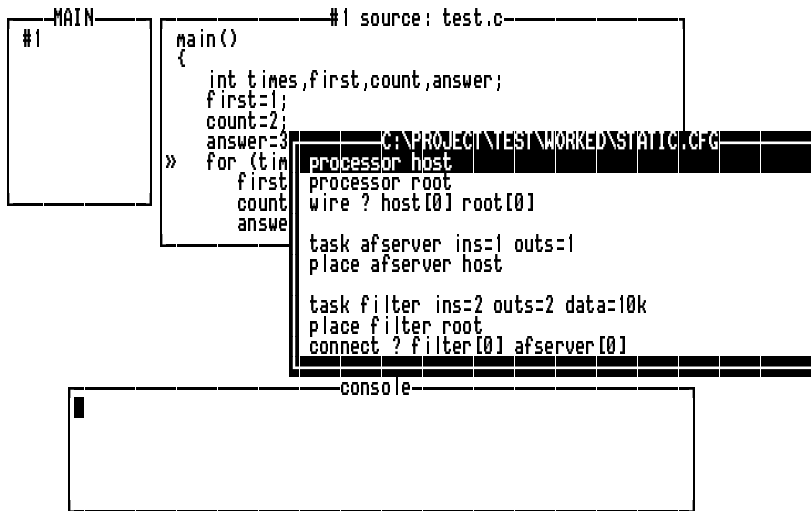


Figure 7.16: A File Window

way to investigate the contents of the components of these.

7.4 File Windows

Tbug will let you examine files during the debugging session. You can create a *file window* by following the procedure described in section 6.2.4, and selecting the **Read a file** option. This will bring up a window with a directory listing. Using the cursor you can select different directories until you have located the file you wish to see. When the cursor is over the file, pressing Enter will dismiss the directory listing and bring up a file window which will display the contents of your file.

Figure 7.16 shows a file window.

You can move about in the file by means of the arrow keys and also PgUp, PgDn, Home and End, as described in section 7.2 for source windows. However, this can be inconvenient if the file is large. If you

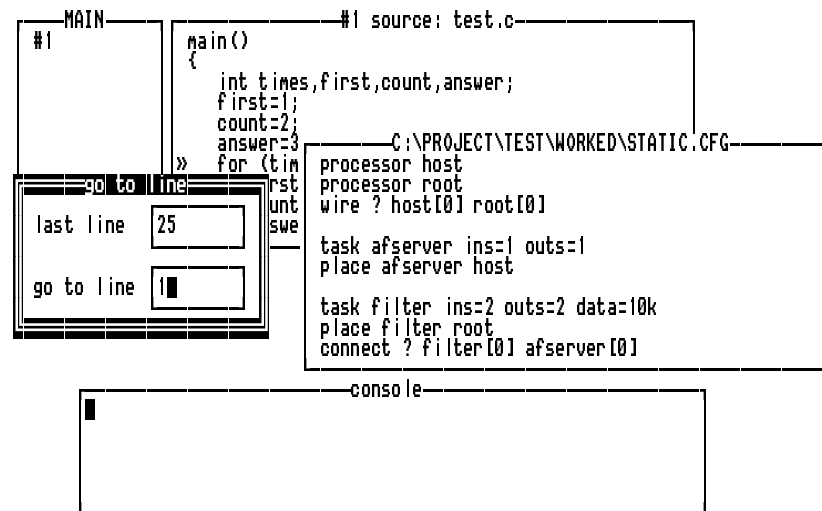


Figure 7.17: A “go to line” Window

press ‘#’ you will get a *go to line* window. By editing in the number of the line you want and pressing Enter you can easily select parts of even large files.

Figure 7.17 shows a “go to line” window.

Chapter 8

Low-Level Features

8.1 Assembler Windows

An *assembler window* displays part of the program you are debugging in assembler mnemonics. It allows you to observe and control the behaviour of your program at the instruction level.

If the program you are working on contains no line-number tables (see chapter 4) Tbug will open an assembler window in place of the normal source window. If you need to create an assembler window at some other time, you should follow the procedure described in section 6.2.4. When an assembler window is created, it is based around the `Iptr` for the thread. The instruction addressed by `Iptr` will be marked with a chevron (») on the left.

Many of the commands you can issue to a source window will have similar effects in an assembler window. For example, you can use the keys `↑`, `↓`, `PgUp`, `PgDn`, `Home` and `End` to move the cursor through the instruction sequence.

Figure 8.1 shows an assembler window.

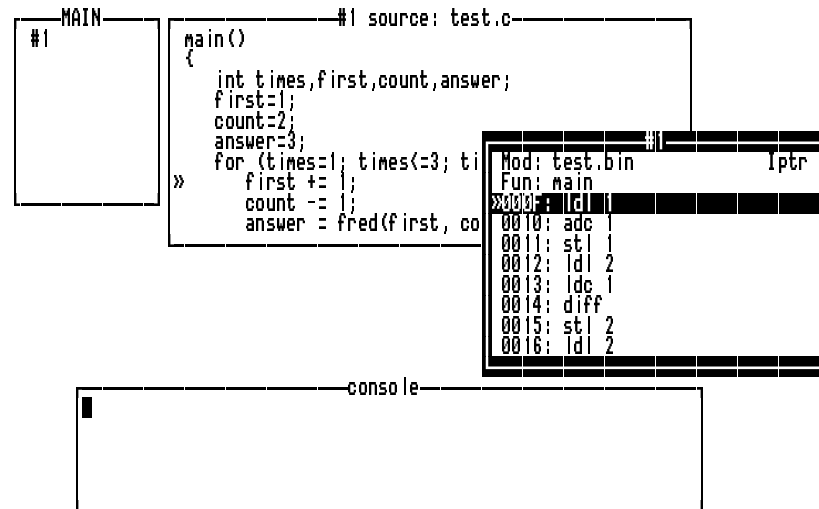


Figure 8.1: An Assembler Window

8.1.1 Locking

When you create an assembler window it will be *locked* to **Ip**tr. This means that while you execute the program, the window will be changed so that the instruction addressed by **Ip**tr, the *current instruction*, will always be visible. This state is indicated by the word **Ip**tr appearing at the top right of the assembler window. When the window is locked, the cursor bar will move to stay on the current instruction. If you move the cursor bar away from the current instruction the window will become unlocked (or *free*). This will be indicated by the word **free** appearing at the top right. A free window will not adjust to follow the current instruction but will keep displaying the selected instructions. You can use ‘L’ to lock the window to **Ip**tr again.

8.1.2 Single-Stepping

When you have selected an assembler window you can single-step the program just as you can in a source window. However, while a source window will step a *statement* at a time, an assembler window will step an *instruction* at a time. Space will step over `call` instructions and Enter will follow `call` instructions and step into the function. Single-stepping inside an assembler window will lock the window to `Iptr`.

8.1.3 Skipping an Instruction

Sometimes you may like to skip the next instruction, that is, move the `Iptr` to the instruction after the one it is currently pointing at. This can be done by pressing ‘S’. This is a speciality of assembler windows, and cannot be used with a source window.

8.1.4 Breakpoints

You can set strong and weak breakpoints on individual instructions using ‘B’ and ‘W’. This is similar to the way breakpoints are set on source lines; see section 7.2.8 for details. You should be careful when working with assembler and source windows at the same time, as breakpoints set on instructions part-way through a high-level language statement can lead to some confusion when single-stepping the source program.

8.1.5 Execute to Here

You can select an instruction and execute up to it using ‘H’ in exactly the same sort of way as for source lines. See section 7.2.4 for details.

8.1.6 Move to Here

Sometimes, you may wish to move the `Iptr` to another place, so that execution will continue from there. This might be done, for example, to skip a whole section of code. The command `Alt-H` will move the `Iptr` to point to the place the cursor is at. Note that quite serious things may go wrong if you misuse `Alt-H`, especially if you attempt to move to another function. `Alt-H` is available only from assembler windows, and source windows do not recognise it.

8.1.7 Following Jumps and Calls

When the cursor is on a jumping instruction, `j`, `cj`, or `call`, you can relocate the assembler window to the instruction referenced by the jump by pressing `F3`.

8.1.8 Backtracking

You can explore the sequence of calls that have led to the current state by means of the '`C`' command. Each time you give this command the assembler window will be repositioned over the `call` instruction that invoked the function containing the current instruction. If you continue the process too far, you will eventually find yourself in the initialisation code used to start your program!

8.1.9 Moving Assembler Windows

An assembler window can be moved to display different parts of a program by selecting the window and pressing `F2`. This will bring up the *assembler window option menu* shown in figure 8.2.

The following options are available:

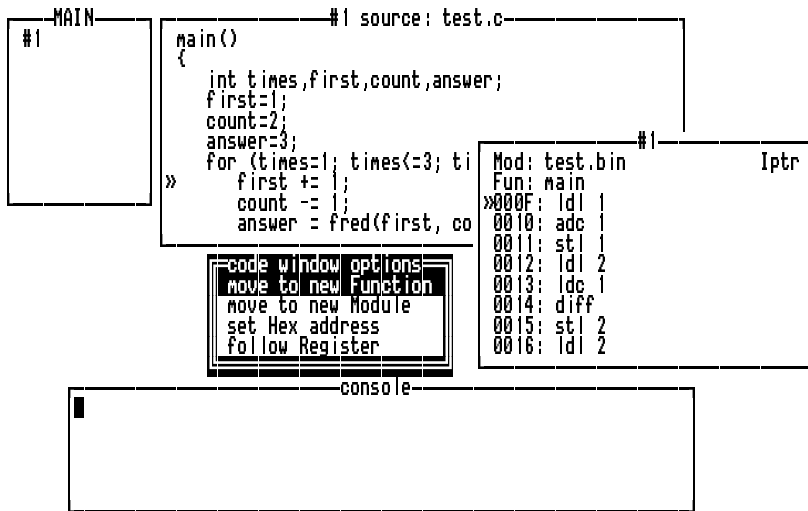


Figure 8.2: Menu for Relocating Assembler Windows

- **move to new Function**

Selecting this option will bring up a *new function* window which will list all of the functions in the current module. If you move the cursor bar onto one of these functions and hit Enter the assembler window will be repositioned over the start of the selected function. Figure 8.3 shows a new function window.

- **move to new Module**

Selecting this option will bring up a *new module* window which will list all of the modules which were linked together to form the program. If you move the cursor bar onto one of these functions and hit Enter you will get a new function window listing the functions in the selected module, as described in the previous item. Figure 8.4 shows a new module window.

- **set Hex address**

Selecting this option will bring up an *hexadecimal editing win-*

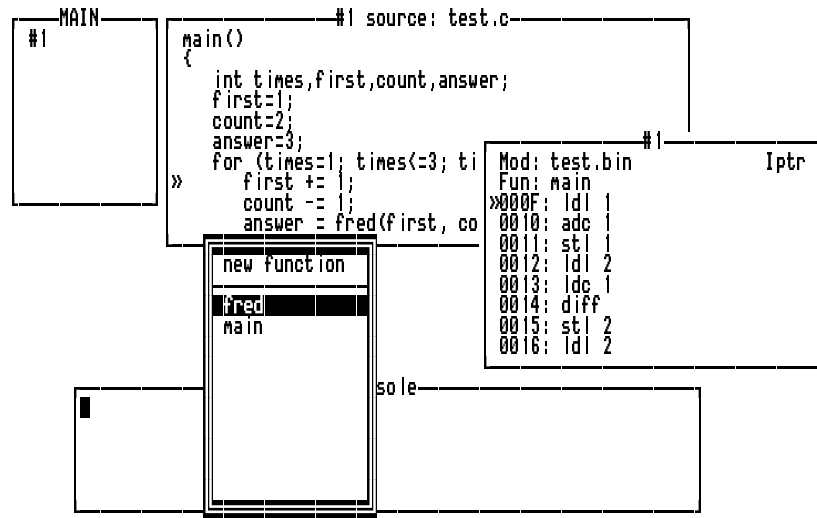


Figure 8.3: New Function Window

dow in which you can specify an absolute address in hexadecimal. Pressing Enter will then move the assembler window to the instructions at that address. Figure 8.5 shows a hexadecimal editing window.

- **follow Register**

Selecting this option will bring up a *register indirection menu* listing the names of the integer registers. If you move the cursor bar onto one of these registers and hit Enter the assembler window will be repositioned over the instruction at the address contained in the selected register. Figure 8.6 shows a register indirection menu.

Esc can be used to abandon this procedure and leave the window in its original position.

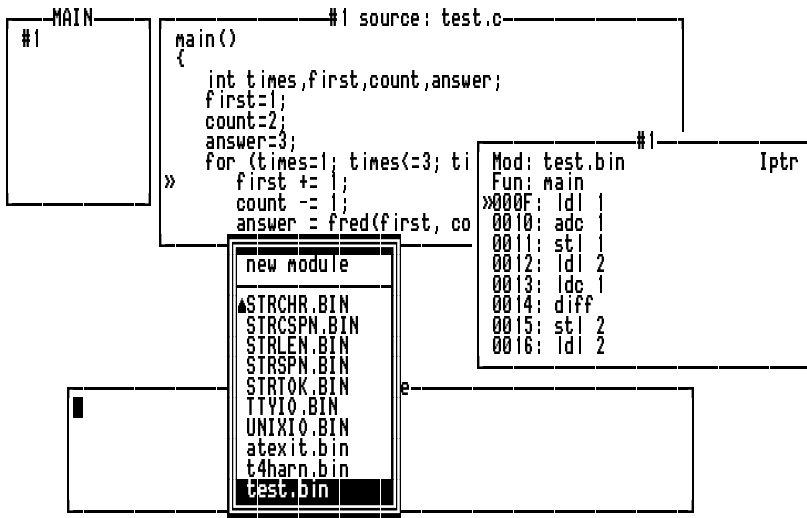


Figure 8.4: New Module Window

8.1.10 Duplicating Assembler Windows

Often you may wish to examine different parts of the program without losing your current place in an assembler window. You can accomplish this by duplicating the window using ‘D’ before moving away. Later, you can delete the new window with Del to return to your original place.

8.2 Register Windows

Register windows are useful when you are single-stepping in an assembler window as they let you observe the detailed results of each instruction as it is executed.

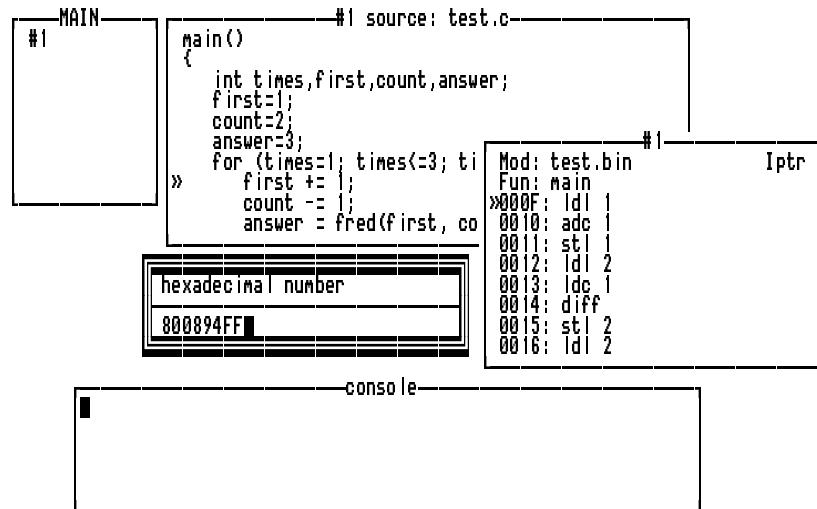


Figure 8.5: Hexadecimal Editing Window

8.2.1 Integer Registers

You can create an *integer register window* by following the procedure described in section 6.2.4. Figure 8.7 shows an integer register window.

The window will show values of the following registers:

- **A, B, C:**

Hexadecimal values of the three evaluation stack registers. The special hexadecimal value 0DEFACED is placed in **Creg** each time the stack is popped to show that its value is left undefined by the transputer hardware.

- **I:**

Hexadecimal value of the instruction pointer, **Iptr**. This contains the address of the next instruction to be executed.

- **W:**

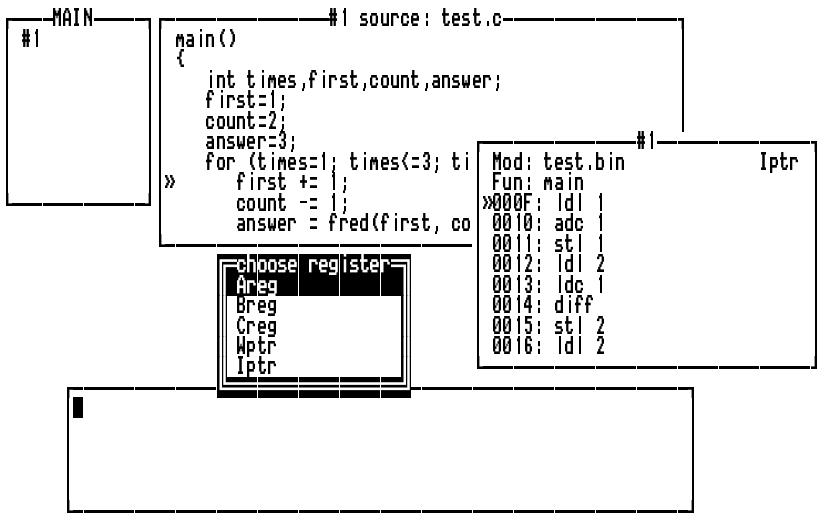


Figure 8.6: Register Indirection Menu

Hexadecimal value of the workspace pointer, Wptr. This is the address used in most memory-referencing computations.

- **E:**
State of the Error flag, Eflag. This can be either On or Off.

8.2.2 Floating-point Registers

You can create a *floating-point register window* by following the procedure described in section 6.2.4. Figure 8.8 shows a floating-point register window.

The window will show values of the following registers:

- **Floating error:**
Error flag for floating-point operations
- **Rounding mode:**

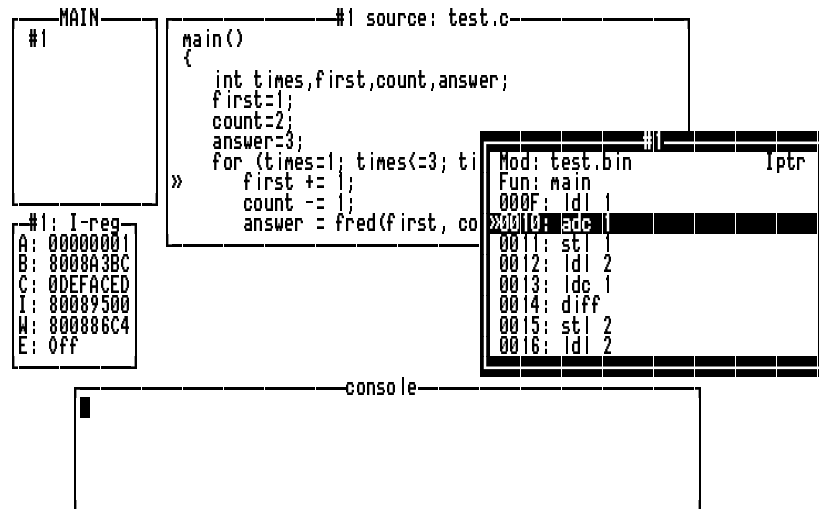


Figure 8.7: An Integer Register Window

One of zero, near, +inf, or -inf

- **FAreg:**

Floating-point A register.

- **FBreg:**

Floating-point B register.

- **FCreg:**

Floating-point C register.

If a register is left undefined by the transputer hardware, Tbug displays its contents as **EMPTY**. Each time the stack is popped, the **FCreg** is left undefined in this way.

A floating-point register window can be created even when Tbug is running on a T414. However, in this case, each of the registers is shown as **EMPTY**, and never changes.

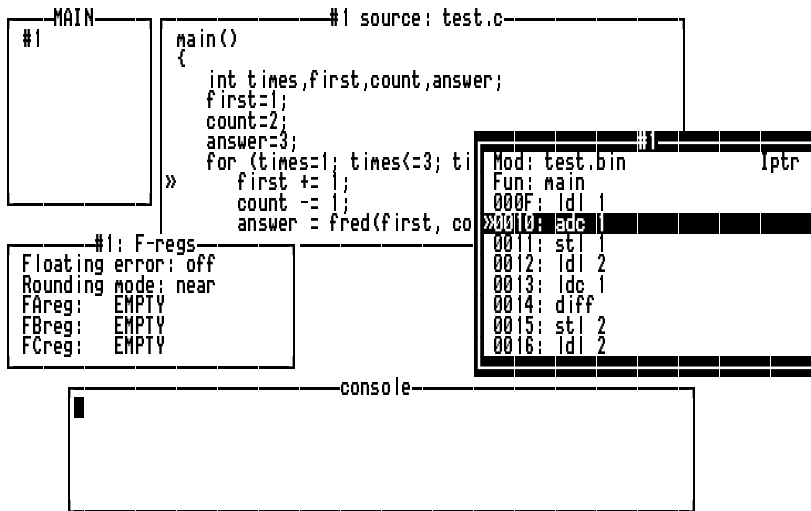


Figure 8.8: A Floating-Point Register Window

8.3 Memory Windows

A *memory window* gives you a view into an area of memory, where you can observe locations being altered, explore data structures, and change stored values.

8.3.1 Creating Memory Windows

8.3.1.1 Based on Wptr

You can create a memory window in the same way as other thread windows, by following the procedure described in section 6.2.4. The area of memory displayed will be based on the value of `Wptr` for the thread associated with the current window; that is, it will show the local stack frame for the currently-executing function of the thread.

Figure 8.9 shows a memory window based on `Wptr`. The numbers in the left-hand column followed by a ‘:’ are hexadecimal addresses.

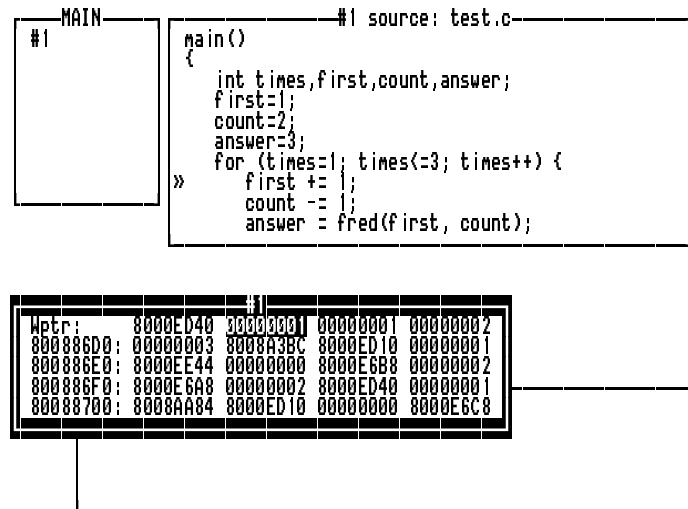


Figure 8.9: A Memory Window Based on Wptr

8.3.1.2 Based on a Variable

You can also create a memory window which will display the values in a memory area based on the address of a program variable. To do this you select your **Variables** window and select a particular variable with the arrow keys. The memory window can then be created by hitting 'M'.

For further discussion of this method of creating memory windows, and the reasons why this facility exists, see section 7.3.6.

8.3.2 Locking

When you create a memory window in the first way described above, it will display the contents of memory locations in the area of **Wptr**. In this state the area of memory being displayed will change each time **Wptr** changes. The window is said to be *locked* to **Wptr**. This is indicated by the symbol "Wptr" appearing at the left of the display

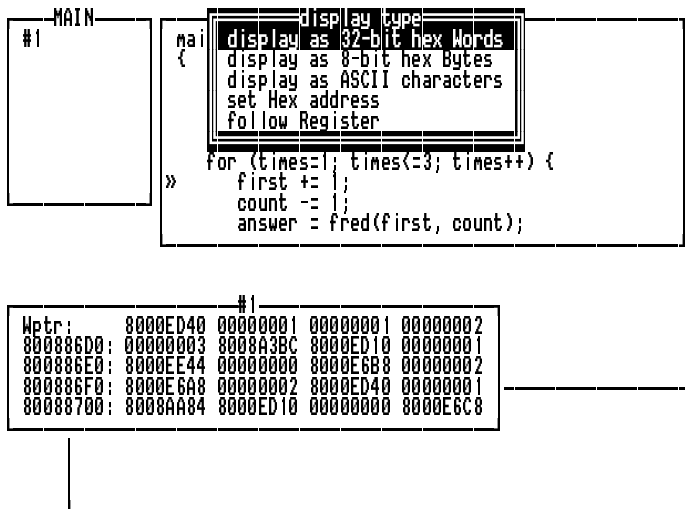


Figure 8.10: A Display Format Window

instead of a memory address. If you select the memory window and move the cursor using the arrow keys then the window becomes unlocked and will not follow changes in the value of `Wptr`. The indicator at the left of the window will be replaced by a memory address to show that the window is unlocked. Any unlocked memory window can be locked to `Wptr` by giving the command 'L'.

8.3.3 Changing the Display Format

A memory window will initially display values as 32-bit integers in hexadecimal. Pressing F2 when a memory window is selected will bring up a *memory window option menu*, shown in figure 8.10, which gives various options as follows:

- **display as 32-bit hex Words**

Display the memory values in hexadecimal as 32-bit integers. This is the default display mode.

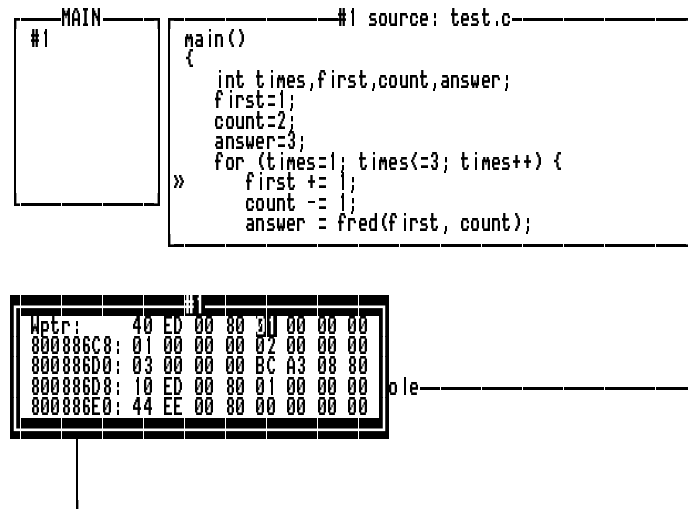


Figure 8.11: Displaying Memory as 8-bit Bytes

- **display as 8-bit hex Bytes**

Display memory values in hexadecimal as 8-bit bytes. Figure 8.11 shows the memory area from figure 8.9 displayed as 8-bit bytes.

- **display as ASCII characters**

Display memory values as ASCII characters. Figure 8.12 shows the memory area from figure 8.9 displayed as ASCII characters.

The other options in this menu are described in section 8.3.4.

8.3.4 Changing the Display Location

When you create a memory window it will initially display an area of memory starting at the address in `Wptr`. You can change the area being displayed in a number of ways:

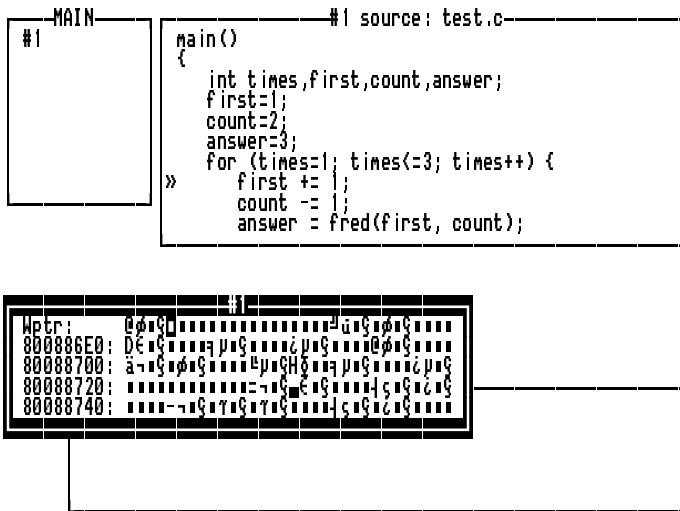


Figure 8.12: Displaying Memory as ASCII Characters

- Use the arrow keys to move the cursor onto new locations. The area being displayed will change to keep the cursor within the window.
- Use PgUp and PgDn to get the effect of repeated uses of the arrow keys.
- Position the cursor over a 32-bit integer value and then make this value the new address for the display by pressing F3. This allows you to follow pointers which are held in locations which cannot currently be accessed as variables, such as pointers within structures.
- Press F2 to bring up the *memory window option menu*, which was described above and shown in figure 8.10. The options include the following for changing the area of memory being displayed:

– set Hex address

Change the address from which the memory display starts. By default the display is based at the value in `Wptr`. An edit window will be created and you can edit in the correct address. Hitting Enter will move the display to the new address. Figure 8.5 shows an edit window.

– **follow Register**

Change the address from which the memory display starts to be a value currently in one of the integer registers. A new menu will be created from which you can select the particular register to use. Figure 8.6 shows a register indirection menu.

8.3.5 Duplicating Memory Windows

When you are exploring memory it can be useful to be able to “remember where you are” while you go off to investigate different memory areas. The easiest way to do this is to *duplicate* the memory window by hitting ‘D’. You can then move the new window around memory without changing the original window. You may make as many copies of a memory window as you wish. These copies can be deleted in the usual way by selecting them with F5 or Alt-F5 and hitting Del.

8.3.6 Changing Memory Values

You can change the values held in memory by interacting with a memory window. The current version of Tbug will only let you change memory one byte at a time. You should therefore change the window to display the memory as 8-bit bytes using F2 as described in section 8.3.3. To change the value of a particular byte you first move the cursor onto the byte in question using the arrow keys and then hit ‘E’. This will bring up an editing window in which you can construct the new byte value. When you leave the editing window by typing Enter the new value will be stored in memory. If you have made a

mistake and do not want to change the memory you can abandon the editing operation by hitting Esc once or twice. Section 6.4.2 describes editing.

Chapter 9

Parallel Execution

Chapters 7 and 8 discussed the various types of thread window. We saw that a thread window is connected to a single thread, and enables you to examine and control some aspect of that thread's execution. This chapter explains how it is possible to use Tbug to debug a program which includes more than one task, and tasks which contain more than one thread, all executing concurrently.

9.1 Multiple Threads

When you debug a simple, non-configured C program you will notice that there is always one window called **MAIN** which contains a single line marked **#1**. This window is a *thread directory*. When Tbug starts a non-configured application, it automatically creates a thread directory as part of its initial display.

The thread directory contains a list of the active threads in the task—in our case, in the single task **MAIN**. In the case of a task with only one thread (such as a normal, sequential C program), this list will contain one entry for the single thread of execution known as **#1**. Thread windows which you create will be connected to that

thread, and will reflect its set of register values, including its `Ip`tr designating where execution is happening, and its `Wp`tr, identifying its own local variables.

If the task uses the `thread_create` or `thread_start` functions to create a new thread, then a new entry is made in the `MAIN` thread directory, say `#2`. By placing the cursor over the `#2` entry and hitting `Ins` you can start to create thread windows connected to the new thread, each of which would contain `#2` in its title. This is discussed further in section 6.2.4.

In general, debugging commands issued to a window will apply only to the thread connected to that window. For example, giving the command `'G'` in a source window will let the thread connected to that window continue running; it will not affect any other threads that have been paused. Obviously, if the thread you have started supplies output that another thread is waiting for, the second thread will start executing again. The same applies to breakpoints. Even though two threads may well be sharing the same instructions, a breakpoint set for one thread will not stop the other.

There is one command provided to act on all threads at once. This is `Alt-G` which has the effect of a `'G'` command on every thread in the application.

9.2 Multiple Tasks

So far, we have only considered single-task, non-configured applications. However, `Tbug` can handle multi-task, configured applications generated by either of the 3L configuration utilities as well.

9.2.1 Starting Configured Applications

The procedure for starting `Tbug` to work on single-task, non-configured applications is described in chapter 5. The procedure

for multi-task, configured applications is different.

When you build an application from several tasks you first *configure* the application using either the General Purpose Configurer or the Flood-Filling Configurer. These 3L utilities take as input a *configuration file* which specifies the component tasks and how they are interconnected. This information will also be needed by Tbug when you wish to debug a multi-task application.

Accordingly, when you invoke Tbug you must supply a configuration file (a `.cfg` file) as the “program” to be debugged and *not* the output from the configurer (commonly an `.app` file).

For example:

```
C>config prog.cfg prog.app
C>tbug prog.cfg
```

Tbug works out the names of the files which hold the various tasks of the application by examining the `.cfg` file. It then looks for them, first in the current directory and then by following the MS-DOS search path, and loads them one by one. If a task file cannot be found, Tbug displays an alert and stops. The `.cfg` file is also needed to make the correct connections between the tasks’ ports.

9.2.2 Working with Tasks

The main difference between single and multiple tasks is that we need to be able to talk about specific tasks, rather than rely on the fact that there is only one. Tbug lets you select tasks by presenting a *task directory* which lists the names of all of the tasks in an application. This task directory forms part of Tbug’s initial display when a configured application is being debugged; you can also create one by selecting the console window and pressing Ins.

When you position the cursor on a task from the task directory and hit Ins a new thread directory will be created. This directory will contain a list of the threads (initially only one) which are active in

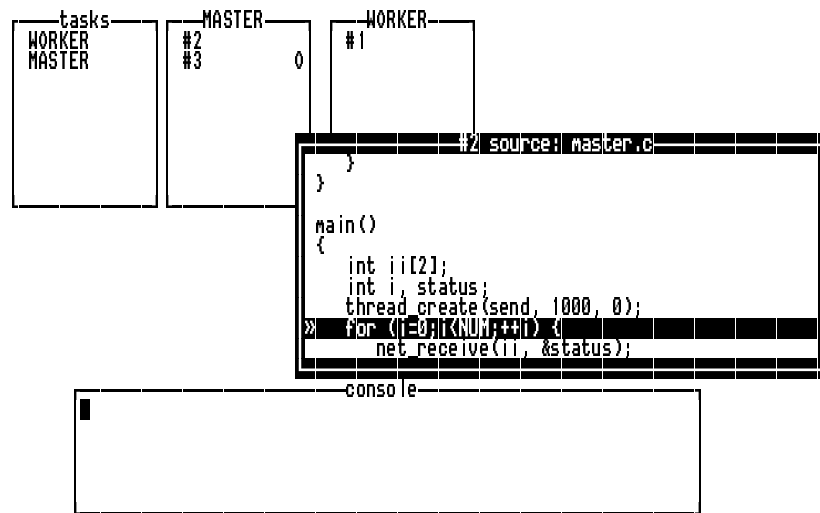


Figure 9.1: Task and Thread Directories

the selected task. If you move its cursor to a thread and hit Ins, you can create a thread window connected to that thread, as discussed in section 9.1 above.

Figure 9.1 shows task and thread directories.

Notice that every thread in the application is given a unique number: there is only one thread #1, not one for every task. This means that the number included in the heading of each thread window is enough to identify both the thread to which it is connected, and the task to which that thread belongs.

Even with a multi-task application, interaction is still with threads one by one. Once again, the Alt-G command may be used to start all threads in all tasks running freely.

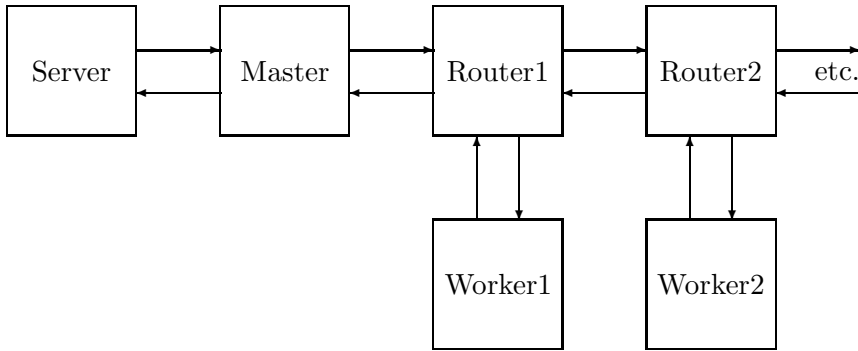


Figure 9.2: Connection of Master and Workers

9.2.3 Flood-Filled Configuration

The configuration file for a flood-filled application specifies two tasks: a **master** task, which is normally located on the host transputer, and a **worker** task, a copy of which is placed on every transputer in the network. A copy of a task called **frouter** is also placed on each transputer; this handles the movement of work and response packets through the network.

As there is normally one copy of the **worker** task per transputer, the number of copies depends on the size of the network. When the application is being debugged, there is only one transputer, and so the number of copies of **worker** to use is set by a preference (see section 11.3); currently, this is 4 by default. The **master** task, the copies of the **worker** task and the necessary copies of the **frouter** task are arranged into a pipeline by Tbug, as shown in figure 9.2.

Once the application has been loaded, Tbug will execute the initialisation code sequences of all the tasks, and then pause each task at

the start of its main function. If you now hit Alt-G the application will run.

9.2.4 General Configuration

Tbug will only handle general-purpose configuration files which fulfill certain conditions. Fortunately, these conditions are satisfied by most applications designed for running on a real transputer network. The conditions are:

1. There must be a processor called **host**;
2. The **host** processor must have a single task which is assumed to be a *server task*, such as the **afserver**;
3. This server task must have one input port and one output port;
4. There must be a *filter task* (usually **filter**) which has exactly two input ports and two output ports, with one pair of ports connected to the server task and the other pair connected to the other tasks in the application.

Tbug loads all the tasks into the root transputer, and establishes connections between their ports as specified in the **.cfg** file. It then executes the initialisation code sequences of all the tasks, and pauses each task at the start of its main function.

The filter task itself must be present in the configuration file although it is not actually loaded or used by Tbug. Filter tasks are described in the 3L Parallel C User Guide[3].

Once again, Alt-G may be used to start all the tasks running.

9.2.5 Choice of Configuration

You do not tell Tbug explicitly whether it is processing a general-purpose or a flood-filled application. This information is deduced

from the statements found in the configuration file. Roughly, if Tbug only finds statements which are supported in flood-filled configuration it will assume that it is a flood-filled application. If any other statements (such as PROCESSOR or WIRE) appear, it will assume a general-purpose application.

Chapter 10

Memory and Link Protection

The default behaviour of Tbug is to prevent the program being debugged from writing to areas of memory which are outside its workspace. The workspace includes the area allocated for the program's stack variables, static variables and heap. Locations outside the program area altogether, or which are occupied by the program's code, are protected in this way. If the program does misbehave in this way, Tbug reports an address error: see section 7.2.6. Note that reading outside the workspace is not prevented.

By default, a special protection is also given to the hardware link locations. As they are outside the workspace, they are protected against being altered; they are also protected against the program performing channel I/O on them.

Tbug provides methods to remove these protections, both for the whole application and thread-by-thread. At the start of a session, all threads have memory and link protection turned on. Threads created while a program is running within Tbug inherit the protection status of the thread which created them.

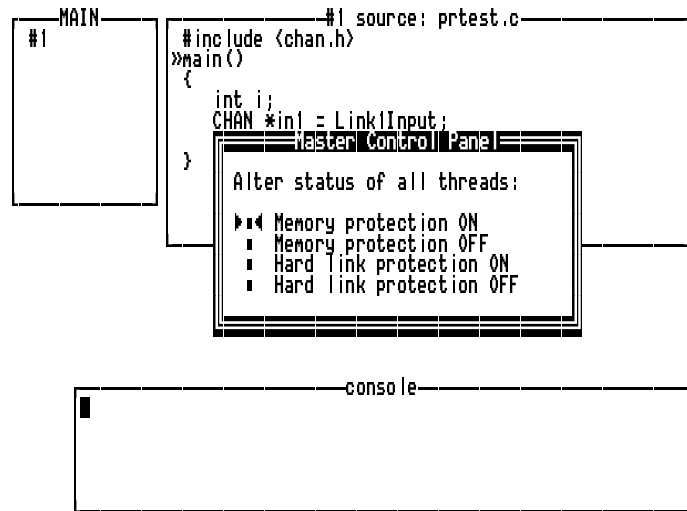


Figure 10.1: Master Control Panel

10.1 The Master Control Panel

The Master Control Panel is the method provided for controlling the memory protection and link protection for the whole application. It can be invoked by using the command Alt-K. Alt-K is a Global Command (see section 6.5) and so may be used whenever an ordinary or help window is selected, but is not recognised when a special window is being displayed.

Figure 10.1 shows the Master Control Panel.

When invoked, this window displays options for turning on or off the memory protection and link protection for all the threads of the application. An option is selected by using the cursor keys to move the indicator to the desired option, and then pressing Enter. Once this has been done, the window is removed and the new status is applied to all threads. You may press Esc to remove the window without making a selection.

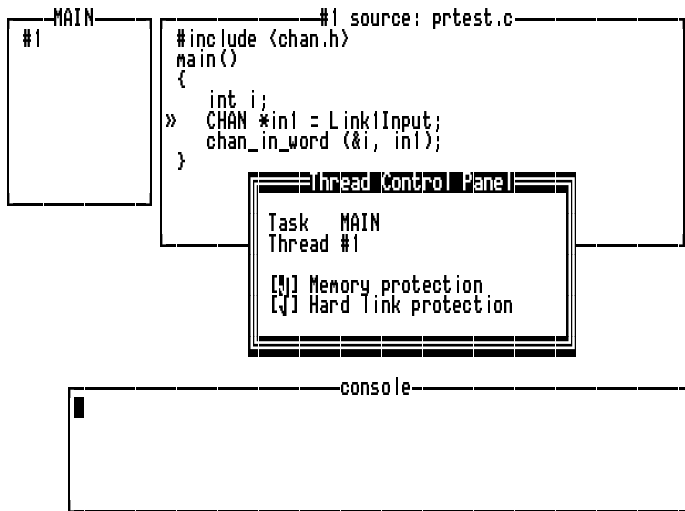


Figure 10.2: Thread Control Panel

10.2 The Thread Control Panel

The Thread Control Panel enables you to control memory protection and link protection for a single thread. It can be invoked by pressing ‘K’ when the currently selected window is a source, variable, assembler, memory or file window which is connected to the thread you wish to affect.

An example of a Thread Control Panel is shown in figure 10.2.

When the Thread Control Panel is invoked, it displays the current settings for memory and link protection for the current thread. When a protection is in force, it will be marked with a ‘v’ symbol. To change the status of a setting, move the cursor between the selectors, using the Tab and Shift-Tab keys. Typing Enter makes the changes effective and removes the window. You can remove the window without making any changes by pressing Esc.

10.3 Disabling Memory Protection

When memory protection is disabled, Tbug allows the program to access areas of memory outside its workspace. The principal use of this facility is not enable the program to use memory-mapped devices on the root transputer, such I/O devices or video RAM. Only devices directly connected to the root transputer are accessible, since Tbug loads all tasks into the root transputer's memory for debugging, and executes them there.

Only the thread dealing with the device need have its memory protection switched off. This means that only this thread is "trusted" and the other threads in the system will still have their memory accesses checked.

10.4 Disabling Link Protection

When the special protection of the hard links is disabled, Tbug allows the program being debugged to perform input and output to a device which is attached to one of the root transputer's links. One might think that this facility could be used, when debugging a multi-transputer application, to communicate with a task which is not on the root transputer. However, it is not intended for this purpose, and the supported method for debugging a multi-transputer application is still to load all the tasks into the root transputer. For this reason, only the hard links of the root transputer are accessible.

Only the thread which deals with the linked device need have its hard link protection disabled. This means that only this thread is "trusted" to do link I/O and the other threads will still be prevented from using the hard links, although they will be allowed to communicate with each other in other ways.

Hard link 0 is reserved for Tbug's own use, and therefore is always protected. Any attempt to perform I/O on link 0 will result in an error (see section 7.2.6.)

The following restrictions in the current implementation should be noted.

- When a thread performs input or output over a hard link, Tbug pauses until the communication is completed. This means that only one thread at a time can perform an input or output operation over a hard link.
- The following should not be used at all on a hard link from Tbug.
 - The `alt` functions.
 - The timeout versions of the I/O functions (`chan_in_word_t`, etc).

Chapter 11

Environment

This chapter describes the environment in which you can debug programs and how you can change certain aspects of the behaviour of Tbug.

11.1 Persistent Data

Many of the features of Tbug that are visible to you as a user are specified in its *persistent data* files. These files include such things as the definitions of windows: their sizes, borders and properties; the help text; and general Tbug parameters. The persistent data files are kept separate from Tbug itself so that you can make changes to them to suit your particular environment or application.

Tbug's two persistent data files are called `basefile.pdm` and `tbug.pdm`. It looks for these files first in the current default directory, and then by searching down the MS-DOS search path.

11.2 WinEdit

WinEdit is a program which allows you to alter the appearance of the windows displayed by Tbug. You can change the characters used to form the window borders, their colour, intensity and background.

WinEdit has its own persistent data file, `winedit.pdm`, and also uses `basefile.pdm`, which is used by Tbug. It is by changing the contents of `basefile.pdm` that WinEdit changes the appearances of windows. WinEdit also looks for its persistent data files first in the current directory, and then by searching down the MS-DOS search path. It is the first copy of `basefile.pdm` that it finds which gets altered.

WinEdit is included on Disk 1 of the distribution kit. It is started with a command of the form:

```
C>winedit
```

You will be presented with a window showing templates of the four types of window that Tbug uses: help windows, alert windows, special windows other than alert windows (dialogs), and ordinary (normal) windows. One of these templates will be selected and displayed with an intensified border. You can change the selected window by using F5.

The borders of the selected window are made up from characters identified by the digits 0–7. If you type one of these digits you will be presented with a menu giving all of the available possibilities for that portion of the border. The arrow keys will move a cursor over this window to pick out the required character. Initially, the cursor will be over the current choice for the character. Pressing Enter will select the character under the cursor and dismiss the menu.

The colours and character styles used in a window are selected by typing one of the letters 'A'–'F'. If you type one of these letters you will be presented with a menu giving all of the available possibilities

for that aspect of the window. You can use the arrow keys to choose a particular style and select it with Enter.

Once you have made the changes you want, you can leave WinEdit by pressing Alt-X. This stores the new window settings in `basefile.pdm`.

11.3 Preferences

Preferences provide a way for you to change the default behaviour of Tbug and to adjust parameters used to control the loading and execution of configured applications.

Tbug stores the preferences in its own persistent data file, `tbug.pdm`. As we noted above, Tbug looks for this file first in the current default directory, and then by searching along the MS-DOS search path. It is the first copy of `tbug.pdm` that Tbug finds which gets altered when you set preferences.

There are two ways of setting preferences: by invoking Tbug with the `/P` command switch, or by giving the command ‘P’ while Tbug is running. If you change the preferences using the first method your debugging session will use the new preferences. If you use the second method and change the preferences while Tbug is running, the changes will not take effect until you next invoke Tbug.

When you ask to change the preferences, Tbug will respond by displaying a *preferences window* containing the current states of the available options. Figure 11.1 shows a preferences window.

You can move the cursor to the next option by pressing Tab, or to the previous option by pressing Shift-Tab; both of these commands ‘wrap round’ to the other end. Some of the options take Boolean values, in which case Space will toggle between the two states: true and false. True is indicated by the symbol ‘√’. The other options take numerical values which can be edited in the usual way (see section 6.4.2).

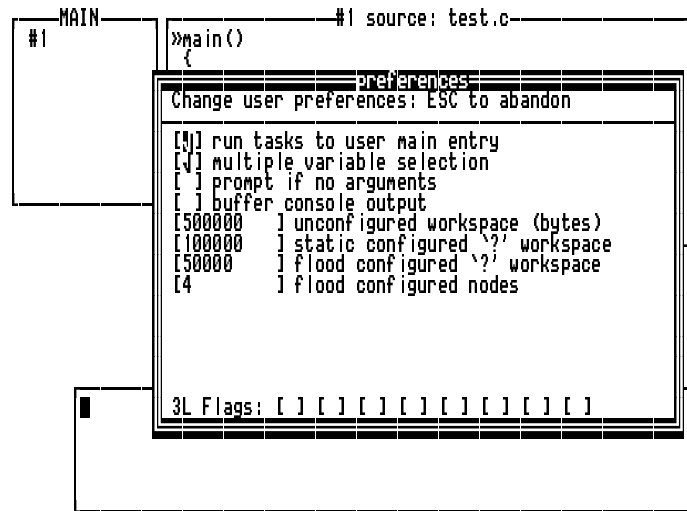


Figure 11.1: The Preferences Window

The options which alter the amount of workspace assigned to the various tasks will, of course, alter the amount of memory required by Tbug; it is not possible to increase these values indefinitely.

The options marked **3L flags** are used by 3L to control certain diagnostic functions. We recommend that users of Tbug should leave them unchanged.

The options currently available are:

- **run tasks to user main entry**

When this Boolean is true, Tbug will automatically execute the system and run-time library initialisation code at the start of a program and pause when it reaches the start of your main program. If you set this Boolean false, then Tbug will pause immediately at the first instruction of the initialisation code.

- **multiple variable selection**

This Boolean controls the behaviour of the variable selection menus after a single variable is selected by pressing Enter (see section 7.3.1). If this Boolean is set (the default), Tbug displays an alert, and when it has been deleted, returns to the variable selection menu. If the Boolean is clear, Tbug simply deletes the variable selection menu.

- **prompt if no arguments**

When this Boolean is true and you have not specified any arguments on your initial command line, Tbug will start by putting up a window asking for a server command line. Whatever you type will be made available to your program as though it had been given on the command line following the name of the program you are debugging. If the Boolean is false, you will not be asked for a command line, even if none had been specified initially.

- **buffer console output**

The phrase “buffer console output” means to save up characters which the program sends to the console until an end-of-line happens, and then display the output all at once. The version 2.0 3L compilers never do this, and so this Boolean has no effect on them. On the other hand, version 2.1 of Parallel C normally does buffer its console output, in order to reduce input/output overheads.

Unfortunately, this buffering is often inconvenient when a program is being debugged. Suppose a program contains the following code:

```
printf("Hello");  
printf(" world");  
printf("\n");
```

It is only when the third line is executed that any output would appear in the console window. For this reason, Tbug by default tries to arrange for output *not* to be buffered. If, however,

you want the program to behave as it would when not being debugged, you should set this Boolean, and output (from C version 2.1 only) will be buffered.

Please note that Tbug will only be able to force tasks to unbuffer their output if they are linked with the `/G` switch (see section 4.2. If a task written in Parallel C version 2.1 is linked without the `/G` switch, its output will be buffered, whether or not this Boolean is set.

- **unconfigured workspace**

This value gives the size (in bytes) of the workspace allocated to an unconfigured application. Changing this preference will affect the size of programs that can be debugged and will also affect the number of windows that Tbug can create.

- **static configured ‘?’ workspace**

The general-purpose configurer takes directions from the user about the allocation of memory to the various tasks and data areas within those tasks. One way in which this can be done is a configuration statement like this:

```
task mytask ... data=?
```

This assigns all the available memory to the task `mytask`. Under Tbug we cannot allow a single task to take all the available memory, as Tbug needs to reserve some for its own use, and for assigning to other tasks which would normally be placed on other processors. So this value defines the size (in bytes) to be assigned when the ‘?’ memory specification is used in a static (that is, a general purpose) configuration file.

- **flood configured ‘?’ workspace**

This value is similar to the previous one, but defines the size (in bytes) to be assigned when the ‘?’ memory specification is used in a flood-fill configuration file.

- **flood configured nodes**

Normally, a flood-filled application contains as many **worker** tasks as there are processors in the network. As we saw in section 9.2.3, when a such an application is debugged, the entire network is placed on Tbug's one processor, so Tbug has to be told how many copies of the **worker** task to load. It takes this number from this value.

Appendix A

Command Summaries

This appendix provides a list of the commands accepted by Tbug along with a brief description of each.

Instead of using the Alt key, you may use the ‘Gold’ prefix key, which is F10. For example, instead of pressing Alt-G to set all threads free running, you could press F10, then press G. This facility is to help those users whose host is an NEC PC-9801, as this computer does not have an Alt key. For further discussion, see appendix B.

A.1 General List of Commands

This summary provides a list of the commands accepted by Tbug, in the following order:

1. Ordinary “character” keys;
2. Function (F) keys;
3. Other keys.

Not all of these commands will be accepted at any one time and some commands may have different meanings in different contexts.

You should consult appendix A.2 below, or the main body of the text if you are in any doubt.

A	Add another variable to a variable window
B	Set a strong breakpoint, or delete a breakpoint
C	Backtrack to caller of current function
D	Duplicate a memory or assembler window
E	Edit the value of a variable or memory location
G	Make the current thread run freely
H	Execute to here
I	Display information about a variable
K	Alter protection settings for this thread
L	Lock a source, memory or assembler window
M	Create a memory window based on a variable
O	Open an array window for this array
P	Change user preferences
S	Skip current instruction
T	Edit the type of a variable
V	Display Tbug's version number
W	Set a weak breakpoint, or clear a breakpoint
-	Break into a running program
#	Goto a specific line or array element
Alt-G	Make all paused threads run freely

Alt-H	Move Iptr to cursor
Alt-K	Alter protection settings for all threads
Alt-X	Terminate the debugging session and return to MS-DOS
F1	Get help relevant to the current window
F2	Select options for a memory or assembler window
F3	Follow a jump or indirect through a memory location
F5	Select the next oldest window
F6	Enter or leave moving window mode
F7	Pop the current window to the front
F8	Push the current window to the back
Alt-F5	Select the next newest window
Shift-F5	Select the next newest window
Backspace	Erase a character when editing; or remove variable from variable window
Del	Delete the current window, or erase a character when editing
End	Move to the end of a file
Enter	Execute a single line or instruction, stepping into functions; or indicate successful completion of an edit or selection operation
Esc	Abandon current operation, or empty an unempty edit box
Home	Move to the beginning of a file

Ins	Create a new window based on the current thread, or follow help reference
PgDn	Move down a few lines
PgUp	Move up a few lines
Space	Execute a single line or instruction, stepping over function calls.
Tab	Select next operation in special window
Shift-Tab	Select previous operation in special window

A.2 Classified List of Commands

This summary is divided into a number of sections. Notice that a command may appear in several sections, either with the same or different meanings.

A.2.1 Controlling Windows

F5	Select the next oldest window
Alt-F5	Select the next newest window
F6	Enter or leave moving-window mode
F7	Bring selected window to front
F8	Send selected window to back
Del	Delete selected window

A.2.2 Cursor Control

The amount of movement performed by some of these commands varies from window to window. If necessary, the contents of the window will be scrolled vertically or horizontally to correspond to the cursor movement.

These commands do not all apply in full to every variety of window.

↑	Move cursor up
↓	Move cursor down
←	Move cursor left
→	Move cursor right
PgDn	Move cursor down a section
PgUp	Move cursor up a section
Home	Move cursor to top of file
End	Move cursor to bottom of file

A.2.3 Commands for Special Windows

There are many different varieties of special window. Only keys common to most of them are listed here.

Del, Esc	Abandon the operation and delete the window
Enter	Indicate selection or input is complete
Tab	Move forwards through operations
Shift-Tab	Move backwards through operations

Also accepted: Cursor Control commands (see A.2.2): some or all are accepted by the various different varieties.

A.2.4 Commands for Help

F1	Invoke help. Doesn't work on special windows or help windows
→, ↓, Tab	Move cursor to next reference
←, ↑, Shift-Tab	Move cursor to previous reference
Home	Move cursor to first reference
End	Move cursor to last reference
Enter	Remove this help window, look up selected reference and display it
Ins	Look up selected reference and display it while retaining the current one
Del, Esc	Remove this help window

A.2.5 Global Commands

These commands are not recognised during the processing of a special window.

P	Enter the preferences menu
V	Display Tbug's version information
Alt-G	Make all paused threads run freely
Alt-K	Alter protection settings for all threads
Alt-X	Terminate the debugging session and return to MS-DOS

A.2.6 Execution and Breakpoint Commands

These commands are mostly available only when a source or assembler window is selected.

G	Go: Make the thread run freely
H	Execute to Here: Make the thread execute freely until it is about to execute the command selected by the cursor
-	Break-in: Stop the thread executing
Space	Single-step: Execute a single line or instruction, stepping over function calls
Enter	Single-step: Execute a single line or instruction, stepping into functions
B	Set strong breakpoint or clear breakpoint
W	Set weak breakpoint or clear breakpoint
Alt-G	(All windows except special windows) Make all threads run freely
S	(Assembler windows only) Skip current instruction
Alt-H	(Assembler windows only) Move <code>Ip_{tr}</code> to the cursor

A.2.7 Commands for Source Windows

Del	Delete the window
Ins	Create new thread window
F1	Invoke help
K	Alter protection settings for this thread

L	Lock the cursor to the instruction pointer
#	Go to a specific numbered line

Also accepted: Cursor Control commands (see A.2.2) and Execution and Breakpoint commands (see A.2.6).

A.2.8 Commands for Variables Windows

Del	Delete the window
Ins	Create new thread window
F1	Invoke help
A	Add another variable to the variable window
E	Edit a variable
I	Display information about a variable
K	Alter protection settings for this thread
M	Open a memory window based on a variable
O	Open an array window for this array
T	Change the type of a variable
Backspace	Delete a variable from the variable window

Also accepted: Cursor Control commands (see A.2.2).

A.2.9 Commands for Array Windows

Del	Delete the window
Ins	Create new thread window

F1	Invoke help
E	Edit a variable
K	Alter protection settings for this thread
#	Go to a specific element

Also accepted: Cursor Control commands (see A.2.2).

A.2.10 Commands for File Windows

Del	Delete the window
Ins	Create new thread window
F1	Invoke help
K	Alter protection settings for this thread
#	Go to a specific numbered line

Also accepted: Cursor Control commands (see A.2.2).

A.2.11 Commands for Assembler Windows

Del	Delete the window
Ins	Create new thread window
F1	Invoke help
F2	Open options menu
F3	Follow <code>call</code> , <code>j</code> or <code>cj</code>
C	Move from function to its calling function
D	Open duplicate assembler window

K	Alter protection settings for this thread
L	Lock the cursor to the Iptr
S	Skip the instruction pointed to by the Iptr
Alt-H	Move the Iptr to point to the location pointed to by the cursor.

Also accepted: Cursor Control commands (see A.2.2) and Execution and Breakpoint commands (see A.2.6).

A.2.12 Commands for Memory Windows

Del	Delete the window
Ins	Create new thread window
F1	Invoke help
F2	Open options menu
F3	Follow pointer
D	Open duplicate memory window
E	Edit current byte
K	Alter protection settings for this thread
L	Lock the cursor to the Wptr

Also accepted: Cursor Control commands (see A.2.2).

A.2.13 Commands for the Task Directory

Del	Delete the window
Ins	Create thread directory

F1 Invoke help

Also accepted: Cursor Control commands (see A.2.2).

A.2.14 Commands for Thread Directories

Del Delete the window

Ins Create new thread window

F1 Invoke help

Also accepted: Cursor Control commands (see A.2.2).

A.2.15 Commands for Console Window

Ins Create a task directory or thread directory

F1 Invoke help

A.2.16 Commands for Edit Boxes

Esc Abandon the editing operation; or empty the edit box

Del Delete the character under the cursor

Backspace Delete the character to the left of the cursor

← Move the cursor one character position to the left

→ Move the cursor one character position to the right

Enter Indicate that the editing operation is complete.

Appendix B

NEC PC-9801 Hosts

Special provision is made in Tbug to handle the facilities provided by the NEC PC-9801 computer when used as a host for a transputer system. Users of IBM PC and compatible computers should disregard this appendix.

B.1 Installation

Users of the NEC PC-9801 should first follow the installation procedure described in chapter 1 in this manual.

Now change to the directory where you have installed Tbug. You will find there a number of files, two of which are **afserver.exe** and **necserve.exe**. The **afserver.exe** program cannot be used to run Tbug on a NEC PC-9801, and it must be replaced by the **necserve.exe** program. This can be done by typing the following command:

```
C>copy necserve.exe afserver.exe
```

Make sure that the IBM version of **afserver.exe** does not appear anywhere in the DOS search path. The new **necserve.exe** program

can be used on the NEC PC-9801 to run any transputer program, including all the compilers, etc.

B.2 The NEC PC-9801 Keyboard

Because the keyboard of the NEC PC-9801 differs from that of the IBM PC in a number of respects, Tbug commands have to be input slightly differently.

- As the NEC PC-9801 keyboard does not have an Alt key, a special key, which we will refer to as the *Gold* key, is used as a prefix to get the same effect. For example, instead of Alt-G, NEC PC-9801 users should press and release the Gold key, followed by the G key.

The Gold key is F10.

- The legends on keys are not all the same as on the IBM PC. This means that in certain cases other keys must be used in place of the keys specified in the rest of this manual. The following table lists the IBM keys which are not available and the NEC PC-9801 keys which should be used instead.

IBM PC Key	NEC PC-9801 Key
Home	HomeClr
End	Shift-HomeClr
PgUp	Roll Up
PgDn	Roll Down
Shift-Tab	F10 then Tab
Alt- <i>key</i>	F10 then <i>key</i>
F1	Help

B.3 The NEC PC-9801 Display

The character set of the NEC PC-9801 differs from that available on an IBM PC. We have tried to make the characters displayed the same on both machines. However, this has not always been possible, and so certain symbols may appear which are not those shown in the examples elsewhere in this manual. These symbols are listed below:

IBM character Appears on NEC PC-9801 screen as

▸	▹
◀	◁
▲	↑
▼	↓
√	*
■	◆
»	}

Where special marker characters differ from those used in the rest of this manual, their meanings should always be obvious from context.

Apart from the characters mentioned above, character codes out-with the range 0...127 may not be displayed as expected. This means that ordinary ASCII values will be displayed correctly, while Katakana will not be displayed as expected. Multi-byte characters (Kanji) are not supported.

Appendix C

Distribution Kit

Tbug is distributed on two 5 $\frac{1}{4}$ " 360KB floppy disks containing the following files:

Disk 1	<code>tbug.b4</code>	Tbug itself
	<code>tbug.exe</code>	driver program used to invoke Tbug
	<code>tbug.pdm</code>	persistent data file used by Tbug
	<code>winedit.b4</code>	WinEdit, the window editing program
	<code>winedit.exe</code>	driver program used to invoke WinEdit
	<code>winedit.pdm</code>	persistent data file used by WinEdit
	<code>basefile.pdm</code>	persistent data file used by both programs
	<code>afserver.exe</code>	server for use on IBM PC and compatibles
	<code>necserve.exe</code>	server for use on NEC PC-9801
Disk 2	<code>test.c</code>	demonstration program
	<code>test.b4</code>	compiled and linked version of <code>test.c</code>
	<code>master.c</code>	demo program for configuration
	<code>master.b4</code>	compiled and linked version of <code>master.c</code>
	<code>worker.c</code>	demo program for configuration
	<code>worker.b4</code>	compiled and linked version of <code>worker.c</code>
	<code>static.cfg</code>	configuration file for general example
	<code>flood.cfg</code>	configuration file for flood-fill example

Bibliography

- [1] *Disk Operating System Version 3.10 Reference*. International Business Machines, February 1985.
- [2] *Microsoft MS-DOS User's Reference*. Microsoft Corporation, 1986. Document Number 410630013-320-R03-0686.
- [3] *Parallel C User Guide*. 3L Ltd, 1989. Software version 2.1.
- [4] *Parallel Fortran User Guide*. 3L Ltd, 1988. Software version 2.0.
- [5] *Parallel Pascal User Guide*. 3L Ltd, 1989. Software version 2.0.
- [6] *Linker User Guide*. 3L Ltd, 1989. Software version 1.0.

Index

- '**W**', *see* execution marker
- '**B**', *see* breakpoints: strong
- '**w**', *see* breakpoints: weak
- '**✓**', *see* preferences
- '**-**', *see* execution: break in
- '**#**', *see* file windows: go to a line;
source windows: go to a
line
- .b4**, *see* executable files
- .cfg**, *see* configuration files