csd™

C Source Debugger

Cuts Development Time
in Half!

Mark Williams Company

V4.0.6 - Rec'd 8/18/87

Mark Williams Company

CSD Registration Form

Please complete this form and return it within 30 days to qualify for maintenance and product updates. This information will help us serve your needs better. Please read and complete all the questions: be specific and check (✓) all items that apply. Your assistance is greatly appreciated.

Customer Data

Name & Title: _____

Company Name: _____

Address: _____

City: _____ State or Country: _____ Zip Code: _____

Daytime Telephone: _____ Extension: _____

Purchased From: _____ Purchase Date: _____

1. How did you hear about this product?

- ☐ Advertisement (magazine): _____
- ☐ Review (magazine): _____
- ☐ Other _____

2. Computer model:

_____

3. Amount of RAM in your computer:

- ☐ 256k ☐ 512k ☐ 640k
- ☐ 1MB ☐ Extended Memory

4. # of floppy disk drives:

- ☐ 1 ☐ 2

5. Do you have a hard disk drive:

- ☐ Yes ☐ No

6. I read the following regularly:

- ☐ Byte ☐ PC Magazine
- ☐ Computer Language ☐ PC Tech Journal
- ☐ Computerworld ☐ PC Week
- ☐ Dr. Dobb's Journal ☐ PC World
- ☐ InfoWorld ☐ Personal Computing

- ☐ Analog ☐ ST Log
- ☐ Antic ☐ START
- ☐ Compute ST ☐ ST Applications

- ☐ Other _____
- ☐ Other _____

7. My level of C programming experience is:

- ☐ beginner ☐ intermediate ☐ professional
- ☐ new to C but professional in other language

8. Describe your current C programming projects:

_____

_____

_____

_____

_____

9. I prefer to purchase software from:

- ☐ retail (full-service) ☐ mail-order
- ☐ retail (discount) ☐ direct from company
- ☐ Other _____

10. This software was purchased for:

- ☐ personal use ☐ business use
- ☐ educational program (please specify)

_____

_____

11. Other software purchase interests (please ✓ items and also elaborate in space provided):

- ☐ Source Level Debugger ☐ C Interpreter
- ☐ C library toolboxes ☐ C tutorial
- ☐ C run-time source code ☐ GEM libraries
- ☐ Cross development tools ☐ Other compilers
- ☐ Other programming tools ☐ Other utilities
- ☐ Embedded applications toolbox

_____

_____

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS • PERMIT NO. 10820 • CHICAGO, IL

POSTAGE WILL BE PAID BY ADDRESSEE

Mark Williams Company
1430 Wrightwood Avenue
Chicago, Illinois 60614

csd™

C Source Debugger

Cuts Development Time
in Half!

Mark Williams Company

Revision 3 Printing 5 4 3 2 1

Published by Mark Williams Company, 1430 W. Wrightwood Avenue, Chicago, Illinois 60614.

I created this PDF version of the *csd C Source Debugger* manual in October 2020 using my 1987 hard copy and archived Mark Williams Company documentation sources. The pages at the beginning and end (cover, Registration Form, title page, User Reaction Report, Other Products, Order Form, Software License Agreement, back cover) are scans of my hard copy. The remaining sections were regenerated from the archived sources using the COHERENT version of **troff**. The reconstructed manual has not been carefully proofread.

Stephen Ness
10/19/2020

Table of Contents

CONTENTS

Introduction

Congratulations on choosing **csd**, the Mark Williams C source debugger. **csd** can speed the development of any C program and make the C language more accessible to the novice programmer and expert alike.

C is a powerful language that combines the flexibility, speed, and compactness of assembly language with higher level language features such as data structures, control structures, and functions. C programs are highly portable and are not tied to a specific machine architecture. Compilers like **Let's C** brought C programming to i8086-based microprocessor systems. Programs that once required assembly language could now be written in C. However, the lack of a true source-level debugger meant that C programs still had to be debugged at machine level.

Now, with **csd**, you can debug your C programs in the same language in which you wrote them. **csd** displays your source code, evaluated expressions, all traced expressions, and program output, each in its own window. Program locations are referenced by their positions in your source file and variables are referenced by name, relieving you of the chore of looking up and calculating numeric machine addresses. If you are a new programmer, **csd** can shorten the time you need to become productive because you no longer need to learn assembly language to debug your programs.

Debugging is the most time-consuming part of program development. **csd** helps make this chore manageable. It is indispensable as a learning tool for newcomers to C, and a time saver for experienced programmers.

Features

csd includes the following features:

- Four different windows let you examine your source code in detail, see your actual program output displayed, evaluate C expressions, and keep a log of traced statements and expressions.
- **csd** lets you enter and evaluate new C expressions, without having to recompile your program.
- With **csd** you can *trace* any executable C statement, stopping execution immediately before the marked line of source.
- You can also trace expressions in the evaluation window. Execution will stop each time an expression marked as traced changes value.
- **csd** gives you a complete list of traced statements and expressions. Each time a program stops at a traced expression, a copy of that line is written to the *history window*, maintaining a complete list of traced statements and expressions.
- On-line help screens make **csd** easy to use, even without a manual.
- You can debug both large and small model C programs with **csd**.

csd does for debugging what high-level languages do for programming: it makes your work easier, saves time, and increases your productivity.

What is csd?

2 Introduction

The task of developing a program can be broken down into several stages, each of which has its own problems.

The first stage is *planning*. At this stage you identify a problem that needs solving, and determine how it can be solved on a computer.

The second stage is *design*. Here, you break the problem into discrete tasks, or *functions*, and sketch out how a program will link the tasks into a whole.

The third stage is *coding*. You translate each function into C code, then compile and link it into an executable program.

The fourth stage is *debugging*. This is the most tedious part of programming. At this stage you methodically test the program, find out where it does not work properly, and fix it. Bugs can result from mistakes at any stage of writing a program: the planning may be faulty, the design may incorrectly interpret the plan, and the coding may have mistakes in it.

Debugging code is often the most difficult and time-consuming part of programming. It is easy to fix a program; the hard part is finding just what is wrong. All experienced programmers can tell "horror stories" about debugging programs that simply do not work. You can study the output to find where the problem might lie. Then you embed **printf** statements within the code to print out the values of certain variables, then recompile and rerun the program; if this does not help, then you embed more **printf** statements elsewhere, recompile, and try again. This continues through perhaps dozens of recompilations until you stumble across the problem. Throughout this process, you find yourself wishing you had a tool that would let you watch the program run so you could stop execution at selected points in the program and examine the values of variables to discover just where the program is going wrong.

csd is such a tool. It lets you peer into your program while it is running. You can run your program from beginning to end, from beginning to any point in the code, or from any point in the code to any other, either all at once or one step at a time. At any point in execution, you can see what the program has output; you can look into memory and see the value of any variable; you can explore the stack and see what functions are calling other functions; and you can enter new C expressions and evaluate them immediately. **csd** does all of this interactively, using windows and keystrokes. You no longer need to alter your program and recompile in order to see how your program is working.

How does csd work?

In essence, **csd** performs two tasks: (1) it starts and stops the execution of a program, and (2) it lets you examine and alter the contents of memory.

Controlling execution

When you invoke **csd**, it loads your program and runs a portion of it. It then saves your *program window* (that is, the screen on which the program writes its output), and displays your source code in the *source window*, with a cursor positioned at the top of the first source file.

You can set a *breakpoint* on any line of source simply by moving the cursor to that line and pressing a key; a breakpoint is a point at which **csd** automatically stops executing the program. A line that bears a breakpoint is said to be *traced*. When you run the program, **csd** starts executing the program from where it last stopped (or from the beginning of the program) to immediately before execution of the next traced line it encounters.

When **csd** stops execution, it saves the program window, retaining what your program has output up to this point. It then redraws the source window, with the cursor positioned at the line of source code at which execution stopped.

You can run your program any number of times while using **csd**.

Referencing data

csd also has an *evaluation window*. There, you may type any C expression that would be valid at your current location in the source window. If the expression is also valid at your current execution position (which may or may not be the same as your current source window position), it is evaluated immediately and its value is displayed. The expression stays in the evaluation window until you delete it.

Since expressions may involve assignment operators (++, --, =, +=, etc.) you can alter your data. The program itself is never altered. Because data alteration may affect the future operation of your program while debugging, this is one way to influence execution. Whenever your program is stopped, every expression in the evaluation window that is valid at the current stopping position is evaluated and its value is displayed.

You can also trace any expression in the evaluation window. **csd** can be instructed to run your program until the value of one of the traced expressions changes.

How to use this manual

This manual introduces you to **csd**. It assumes that you are familiar with the C programming language. ", as well as with DOS and its commands."

This manual also contains everything you need to use **csd** to its fullest advantage. It is organized into the following sections:

1. *Introduction*.
2. *Becoming familiar with* **csd**. This tutorial walks you through the sample C program,**infl**, to demonstrate **csd**'s basic functions.
3. *Advanced features*. Here, the tutorial walks through a more complex program, **factor**, to demonstrate advanced **csd** features. The use of both tracepoints and the evaluation window are two of the features described in greater detail.
4. *A sample debugging session*. This section uses the program **inflbug** to demonstrate how you can use **csd** to track down some of the more common C programming bugs.
5. *Questions and answers*. This section presents a number of questions that new users commonly ask about **csd**. If you have a question about **csd** or its use, look here first.
6. *Commands reference*. This section summarizes all of **csd**'s commands.
7. *Help screens*. This section reproduces all of **csd**'s help screens. It can also be used as a quick reference.
8. *Error messages*. This last section presents all of **csd**'s error messages, discusses what each one means, and gives hints on how to address the problem.

Conventions used in this manual

This manual represents the cursor by a block character '▮'. On your screen, of course, it will be a flashing underscore or block.

Text that is highlighted on your screen is represented in this manual by shaded print.

Sections 2 through 4 of this manual are tutorials on using **csd**. They also reproduce a number of sample screens. As you work through the tutorial, the displays on your computer may differ slightly in other ways from those shown in this manual. For example, you may be logged into a different drive, so it may say **A>** rather than **C>**. These differences should be minor, and will not affect how

csd operates on your computer.

User registration and reaction report

Before you go any further, please fill out the User Registration Card that came with your copy of **csd**. Returning this card will make you eligible for direct telephone support from the Mark Williams technical staff. Also, if you have comments or reactions to the **csd** software or documentation, please fill out and mail the User Reaction Report included at the end of the manual. We especially wish to know if you found errors in this manual. Mark Williams Company needs your comments to continue to improve **csd**.

Installing csd

The following files are on the **csd** distribution disk:

CSDXL.OBJ	CSDSELEC.HLP
CSDXS.OBJ	CSDTRACE.HLP
CSD.EXE	ATOD.C
LCSD.EXE	FACTOR.C
CSDEVAL.HLP	FACTOR.EXE
CSDEDIT.HLP	INFL.C
CSDFIND.HLP	INFL.EXE
CSDHELP.HLP	CC4.EXE
CSDRUN.HLP	INSTALL.DAT
INSTALL.EXE	INFLBUG.C

Before you begin to use **csd**, be sure to make a backup copy of the distribution disk. *Never work with the distribution disk: always work with a copy.* When you have made your backup copy, put the distribution disk away in a safe place.

Installing **csd** involves copying files from the distribution disk onto either a hard disk or a floppy disk. To use **csd** with a two-floppy system, simply copy the distribution disk to a formatted disk.

To copy **csd** to a hard disk, use **csd**'s **install** utility, which does the copying for you. By running **install** and answering a few simple questions, you can build a working copy of **csd** on your hard-disk system in a few minutes.

Installing csd onto a hard disk

To begin, log onto drive C on your system. On nearly all computers, this is the hard disk. Then insert the **csd** distribution disk into floppy drive A and type the following command:

```
a:install
```

In a moment, **install** will begin to work. It will print some information on your screen, and then ask you the following question:

```
Do you wish to install all the files?
```

Type 'y' for **yes**. **install** will now ask you in which directories you wish to install the files, as follows:

```
Where do you want executable programs(default: "\bin")?
Where do you want libraries (default: "\lib")?
Where do you want sample programs (default: "\sample")?
```

After each question, type **<return>**, which accepts the default setting. Later you may wish to re-install **csd** into other directories of your own choice, but at present it is best to use the default settings.

install will now begin to copy the files from the distribution disk onto your hard disk.

csd C source debugger

That's all there is to it. **csd** is now installed on your hard disk.

How to run csd

To debug a program with **csd**, you must first compile it with the **Let's C** compiler using the **-VCSD** option to the **cc** command line. For example, the sample program **infl**, which is included on your **csd** distribution disk, was compiled using the command line:

```
cc -f -VCSD infl.c
```

If you are using the Mark Williams Shell, MWS, be sure to select the **CSD** variant option when you compile. For complete information on compile command line options and MWS, see the **Let's C** manual.

Invoking csd through the Mark Williams Shell

It is recommended that you use **csd** through MWS, the Mark Williams Shell. The shell contains a disk accelerator that will speed up **csd** noticeably. In addition, MWS gives you two ways to enter **csd** commands: either through its graphics interface, which helps you build command lines, or directly through **command.com**.

To use **csd** through the MWS graphics interface, do the following: First, invoke MWS by inserting the **shell** disk from **Let's C** into drive A (if you have a two-floppy disk system) and typing MWS. Remove the **shell** disk from drive A and insert your backup copy of the **csd** disk.

Now, press the down-arrow key <↓> until the reverse-video band, called the *cursor bar*, covers the **Debug** on the main menu. Press **<return>**. MWS will draw a new screen for you; this screen gives you access to all of **csd**'s options and features.

Press the <↓> key until the cursor bar is at **Files**; press **<return>**. A new box will open on the screen; this box displays all of the executable files that are available in the current directory. Press the <↓> key again until the cursor bar covers the entry **infl.exe**; press **<return>**. As you can see, the name **infl.exe** has been written into the command box at the top of the screen and the cursor bar has returned to the entry **Execute** on the main **Debug** menu. Now, press **<return>** again. This executes **csd** with the program **infl.exe**

If you prefer not to use the graphics interface, do the following. First, invoke MWS as described above. Then remove the **shell** disk and insert the **csd** disk into drive A. When MWS's main menu appears, press the <↓> key until the cursor bar covers the entry **!DOS Escape**. Press **<return>**. The screen will clear, and your normal DOS prompt will appear. Now you can type commands directly into DOS; however, MWS will still be working in the background to accelerate your programs. To return to MWS, simply type **exit**, and the MWS main menu will reappear.

To invoke **csd** now, simply type

```
csd infl
```

at the DOS prompt.

Using csd

Once you have installed **csd**, try running the program on the sample program **infl**. **infl** is already compiled with the appropriate command line options, and is ready to debug.

To invoke **csd**, type the following command from the MS-DOS prompt:

```
csd infl
```

csd will load your program and display the beginning of the source code on your screen. Your screen should appear as follows:

```
#include <stdio.h>
main()
{
    int i;                          /* count ten years */
    float w1, w2, w3; /* three inflated quantities */
    char *msg = " %2d\t%f %f %f\n"; /* printf string */

    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++) {  /* apply inflation */
        w1 *= 1.07;
        w2 *= 1.08;
        w3 *= 1.10;
        printf (msg, i, w1, w2, w3);
```

infl.c

To make **csd** run the program, type the *run* key, **<F4>**, followed by the *end* key, <End> . The *program window*, where your program writes its output, will appear briefly as **infl.exe** executes; then **csd** displays the source window with your program's source code.

Now, press the *program* key, **<F7>**. **csd** redisplays the program window so you can study the output of your program. You will see the following screen:

```
C>csd infl
C Source Language Debugger Version 1.1
Copyright (c) 1984-1988 by Mark Williams Company, Lake Bluff, IL
Reading source...
Loading...
  1   1.070000 1.080000 1.100000
  2   1.144900 1.166400 1.210000
  3   1.225043 1.259712 1.331000
  4   1.310796 1.360489 1.464100
  5   1.402551 1.469328 1.610510
  6   1.500730 1.586874 1.771560
  7   1.605781 1.713824 1.948716
  8   1.718186 1.850930 2.143588
  9   1.838458 1.999004 2.357946
 10   1.967150 2.158924 2.593741
```

Now, type the *source* key, **<F8>.** The screen will again display the source code.

It is this easy to use **csd**: one command invokes the debugger and displays your source code, two keystrokes execute your program, one keystroke displays the results for you and another returns you to the source code to continue debugging.

Now, type **<Shift-F1>**, the *exit* key, to exit **csd**.

Where to go from here

If you are a newcomer to C programming, or have never used a source level debugger, you should work through sections 2 through 4 to learn how to use **csd**. Experienced programmers will want to go directly to sections 6 through 8. The *Commands reference* section, along with **csd**'s on-line help screens, should be all you need to get started with **csd**.

Becoming familiar with csd

This section walks through the sample C program **infl**, which is provided on your **csd** distribution disk. It demonstrates **csd**'s basic features, such as how to move the cursor, shift windows, and execute a program under **csd**.

infl is a simple program with a **for** loop. It calculates three different rates of inflation over a span of ten years. The source code is as follows:

```
#include <stdio.h>
main()
{
        int i;          /* count ten years */
        float w1, w2, w3; /* three inflated quantities */
        char *msg = " %2d\t %f %f %f\n"; /* printf string */
        i = 0;
        w1 = 1.0;
        w2 = 1.0;
        w3 = 1.0;
        for (i = 1; i <= 10; i++) {   /* apply inflation */
                w1 *= 1.07;
                w2 *= 1.08;
                w3 *= 1.10;
                printf (msg, i, w1, w2, w3);
        }
}
```

This program has already been compiled for you. To see its output, type

```
infl <return>
```

at the MS-DOS prompt. Note that in this tutorial, each line that you type must be followed by **<return>**, unless the text specifically instructs you differently.

Your screen will show the following:

```
C>infl
  1   1.070000 1.080000 1.100000
  2   1.144900 1.166400 1.210000
  3   1.225043 1.259712 1.331000
  4   1.310796 1.360489 1.464100
  5   1.402551 1.469328 1.610510
  6   1.500730 1.586874 1.771560
  7   1.605781 1.713824 1.948716
  8   1.718186 1.850930 2.143588
  9   1.838458 1.999004 2.357946
 10   1.967150 2.158924 2.593741

C>
```

Function keys and what they do

You can manipulate **csd** through the function keys on your keyboard and with the keys on the numeric keypad. These keys let you move the cursor, page through your source code, and move to the beginning or end of your source code.

While using the Mark Williams C compiler, **Let's C**, you have probably become familiar with the editor, MicroEMACS. Switching between MicroEMACS and **csd** while debugging and recompiling doesn't require that you remember two sets of keystrokes. The commands you use to move through text with MicroEMACS will also work with **csd**.

Below is a quick reference list which summarizes all of the command keys and their functions.

FUNCTION	DESCRIPTION	COMMAND
Find	Find a string of text	<F1> or <esc>1 or <ctrl-S> or <esc>S or <ctrl-R> or <esc>R
Select	Select an option	<F2> or <esc>2
Trace	Trace an expression	<F3> or <esc>3
Run	Run the program	<F4> or <esc>4
Cancel	Cancel the last command	<F5> or <esc>5 or <F5>
Help	Display a help screen	<F6> or <esc>6 or <F6>
Program	Display program window	<F7> or <esc>7
Source	Display source window	<F8> or <esc>8
Evaluation	Enter evaluation window	<F9> or <esc>9
History	Display history window	<F10> or <esc>0
Up	Move cursor up	<↑> or <ctrl-P>
Down	Move cursor down	<↓> or <ctrl-N>
Out	Move to calling function	<←> or <ctrl-B>
In	Undo an **Out**	<→> or <ctrl-F>
Exit	Exit **csd**	<Shift-F1> or <ctrl-X> <ctrl-C>
Current	Return to current line	<Shift-F8> or <ctrl-X>X
Page Up	Move cursor up a page	<ctrl- ↑> or <esc>V
Page Down	Move cursor down a page	<ctrl- ↓> or <ctrl-V>
Delete	Delete a line	<Del> or <ctrl-K>
Insert	Insert a line	<Ins> or <ctrl-O>
Beginning	Beginning of source	<ctrl- ←> or <esc><
End	End of source	<ctrl- →> or <esc>>

See section 5, *Commands reference*, for a complete listing of all the function and keypad keys, the corresponding MicroEMACS command keys, and alternate keystrokes used to control **csd**.

Running csd

Now, try running **infl** under **csd**. If you are using the graphics interface to MWS, invoke **csd** on the program as described in the Introduction.

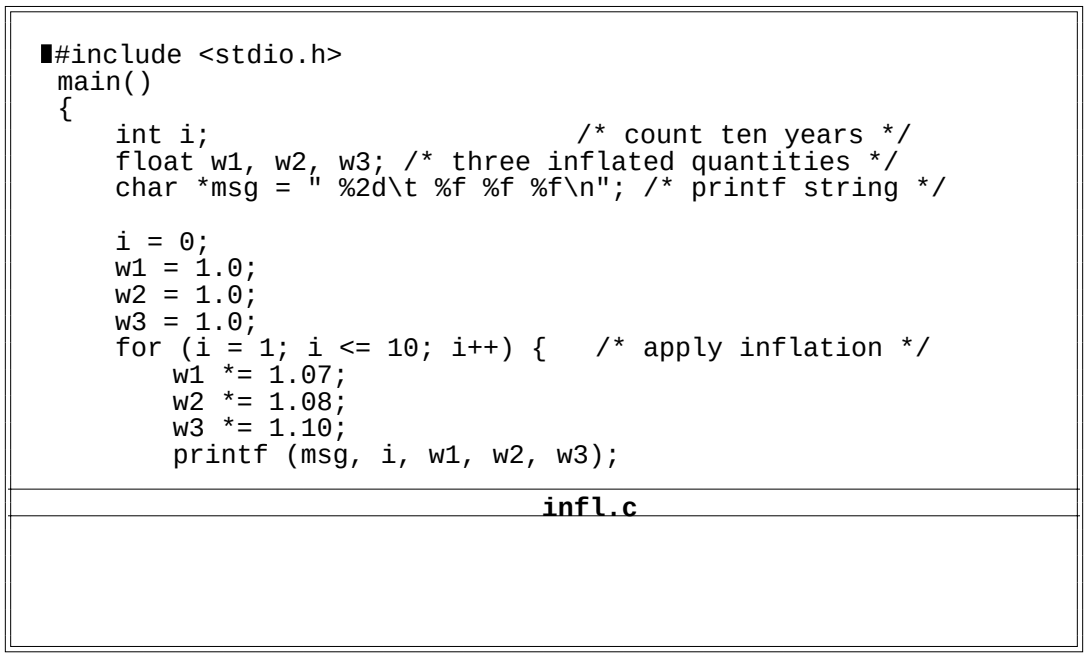
If, however, you are using the **!DOS Escape** option, simply type the following at the prompt:

```
csd infl
```

In a moment, **csd** will load your program and display the beginning of the source code on your screen.

The screen is divided into two parts. The first and largest of these parts is the *source window*, the top portion of the screen where the source code is displayed. The reverse video line separates the two windows and displays the name of the program module you are currently debugging. Under the reverse video line is the *evaluation window*; it is now empty.

Your screen appears as follows:



The *program window* displays the output of the program you are running. The program window does not appear at the same time as the source and evaluation windows; when you display the program window, the entire screen will be redrawn temporarily.

To see the program window, type the *program* key, **<F7>**. Your screen now shows the following:

```
C>infl
  1   1.070000 1.080000 1.100000
  2   1.144900 1.166400 1.210000
  3   1.225043 1.259712 1.331000
  4   1.310796 1.360489 1.464100
  5   1.402551 1.469328 1.610510
  6   1.500730 1.586874 1.771560
  7   1.605781 1.713824 1.948716
  8   1.718186 1.850930 2.143588
  9   1.838458 1.999004 2.357946
 10   1.967150 2.158924 2.593741

C>csd infl
C Source Language Debugger Version 1.1
Copyright 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading source...
Loading..
```

To redisplay the source window, press the *source* key, **<F8>.** The source window is now restored.

Moving through the source code

You can move **csd**'s cursor by pressing the cursor movement keys, which are found on the numeric keypad on the right side of your keyboard. If, instead of moving the cursor, numbers appear on the screen when you press any of the keys on the numeric keypad, press the **NumLock** key. This should solve the problem. .PP To move the cursor *down* on the screen, press the <↓> key. Try it. As you can see, the cursor is now positioned at the beginning of the second line of the source code. If you hold the key down, the cursor scrolls down the screen. The MicroEmacs command **<ctrl-N>** also moves the cursor down the screen one line at a time.

To move back *up* the screen, press the <↑> key. As you can see, the cursor has moved to the previous line. Holding this key down scrolls the cursor up the screen. Typing the MicroEMACS command **<ctrl-P>** also moves the cursor up the screen one line at a time.

Note that if you try to move the cursor past the beginning or the end of your source file, **csd** prints the error message **try Help** at the bottom of your screen.

If you wish to move the cursor more than a few lines, it is handier to use the *page* keys <PgUp> and <PgDn> . These keys move through your source program a page at a time. A *page* is one window of lines. Thus, if your source window displays 19 lines of text, as it does when **csd** starts up, then pressing <PgDn> displays the next 19 lines of text (unless, of course, you are at the end of the source file). When you type <PgUp> , the previous 19 lines of text are displayed. If you are within 19 lines of the end of the source file, <PgDn> will move the cursor to the end of your file; likewise, if you are within 19 lines of the beginning of your source file, <PgDn> will move the cursor to the beginning of the file.

The MicroEMACS **<ctrl-V>** command will also move the cursor to the next page of lines; **<esc> V** moves the cursor to the previous page of lines.

To move the cursor to the end of your source file, type <End> . Try it. The cursor is now positioned at the end of the last line of **infl.c**. To return to the beginning of your source code, type <Home> Try it. The cursor is again positioned at the beginning of **infl.c**.

<ctrl-A> and **<ctrl-E>** move the cursor to the beginning and end of the source, respectively.

Finding text

If you want to locate a specific line in your program, you could hunt for it by scrolling through the text line by line or page by page. However, to ease the search, **csd** has a string search feature. With the *find* key, **<F1>**, you can type in a string that **csd** will find for you.

To see how this works, type **<F1>**. At the bottom of the screen, **csd** prints the following prompt:

```
find:[pattern] ↑ ↓ <Home> <End> Cancel<F5> Help<F6>
```

To find the string **main** in **infl.c**, type

```
main
```

followed by **<return>** or **<↓>**. The screen now displays the following:

```
main()
 {
     int i;                          /* count ten years */
     float w1, w2, w3; /* three inflated quantities */
     char *msg = " %2d\t %f %f %f\n"; /* printf string */

     i = 0;
     w1 = 1.0;
     w2 = 1.0;
     w3 = 1.0;
     for (i = 1; i <= 10; i++) {  /* apply inflation */
         w1 *= 1.07;
         w2 *= 1.08;
```

infl.c

You can control the direction of search with the <↑> and <↓> keys. The <↑> key searches for the string above the current position of the cursor, and the <↓> key searches below it.

The MicroEMACS commands **<ctrl-S>** and **<ctrl-R>** will also search for a string. You can also use these commands to search for strings in your source code:

<esc>S
<esc>R
<esc>l

Exiting from csd

If you wish to exit from **csd**, press **<Shift-F1>**. **csd** will exit and restore the program window. You can use the **<Shift-F1>** key to exit from **csd** at any time during debugging. However, if you are in the middle of typing something in the evaluation window, you must type **<ctrl-U>** before you exit **csd**. Type **<Shift-F1>**. You must also exit the help screens before trying to exit **csd**.

Setting tracepoints

As noted in section 1, **csd** lets you set *tracepoints* within a program to control its execution. When you set a tracepoint and execute a program, the program stops executing at the tracepoint. You can examine variables and see what has happened to them up to that point in the program. This allows you to run a program step by step, so you discover more easily the point at which your program fails.

To see how you can set tracepoints, invoke csd on the program **infl** by typing:

```
csd infl
```

Using the arrow keys, or the MicroEMACS cursor control keys, position the cursor at the first executable statement after the **for** statement:

```
w1 *=1.07;
```

Press the *trace* key, **<F3>**; pressing **<F3>** will set a tracepoint on the line where the cursor is positioned. As you can see, setting a tracepoint on a line of code causes that line to be highlighted on your screen. Your screen should now look like this:

```
#include <stdio.h>
main()
{
    int i;                          /* count ten years */
    float w1, w2, w3; /* three inflated quantities */
    char *msg = " %2d\t %f %f %f\n"; /* printf string */

    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++) {  /* apply inflation */
        w1 *= 1.07;
        w2 *= 1.08;
        w3 *= 1.10;
        printf (msg, i, w1, w2, w3);
```

infl.c

If you were to type **<F3>** again, the tracepoint would be removed and the line would return to its normal intensity. For now, leave the tracepoint on.

Executing to the tracepoint

Now, run the sample program to the traced line. To do so, press **<F4>** to run, and **<F3>** to trace.

This combination of keys tells **csd** to execute your program to the *current statement*. The current statement is the next line of code to be executed. In this example, the program began execution at its beginning, and continued to execute until it encountered the tracepoint you set a few moments ago.

Note that the cursor is again positioned where you set the tracepoint.

Again, type **<F4>**and **<F3>. csd** runs until it again encounters the traced line. While it is executing the program to the tracepoint, it will temporarily display the program window. When **csd** has executed to the tracepoint, it restores the source window.

To see the program's output, press **<F7>.** The screen will show the following:

```
      4 1.310796 1.360489 1.464100
   5   1.402551 1.469328 1.610510
   6   1.500730 1.586874 1.771560
   7   1.605781 1.713824 1.948716
   8   1.718186 1.850930 2.143588
   9   1.838458 1.999004 2.357946
  10   1.967150 2.158924 2.593741

C> csd infl
C Source Language Debugger Version 1.1
Copyright 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading source..
Loading...

C> csd infl
C Source Language Debugger Version 1.1
Copyright 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading source...
Loading...
 1    1.070000 1.080000 1.100000
```

Because the tracepoint is set within the **for** loop, you can see what the program generates with each iteration.

Return to the source window by typing **<F8>**. Press the trace key **<F3>**; this removes the tracepoint. The line returns to normal intensity, and the cursor remains at that line.

History window

Every time your program stops at a tracepoint, **csd** writes a copy of the traced line into the *history window*. This window is where **csd** logs all the statements that have been traced during the execution of your program.

To see the history window, type the *history* key, **<F10>.** If you have followed the steps of the tutorial so far, your screen will look like this:

```
w1 *= 1.07;
w1 *= 1.07;
```

The same statement appears twice because the program stopped executing there twice.

Type **<F8>** to return to the source window.

Single-stepping through a program

With **csd**, you can step through a program a line at a time and examine its execution in detail. This procedure is called *single-stepping*.

To see how single-stepping works, you should first restart your program. You need to do this because if you have been following the steps of this tutorial, the program is midway through its execution. To restart a program, type **<F4>** followed by the *begin* key, <Home> . **csd** reloads the program, then returns the cursor to the beginning of the source file. Your screen appears as follows:

```
#include <stdio.h>
main()
{
    int i;                          /* count ten years */
    float w1, w2, w3; /* three inflated quantities */
    char *msg = " %2d\t %f %f %f\n"; /* printf string */

    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++) {
        w1 *= 1.07;
        w2 *= 1.08;
        w3 *= 1.10;
        printf (msg, i, w1, w2, w3);
```

infl.c

Now, to execute your program one step at a time, press **<F4>** followed by **<return>**. The program window (which shows the program's output) is displayed briefly; then, the source window is restored, with the cursor positioned at the first executable statement in the program:

```
i = 0;
```

Again, press **<F4>** followed by **<return>**. **csd** runs to the next executable line in the program, which is:

```
w1 = 1.0;
```

Now, single-step through **infl.c**, pressing **<F4>** and **<return>**; stop after the first iteration of the **for** loop: your cursor will be positioned at the line

```
w1 *= 1.07;
```

for the second time since you began single-stepping. This is the point at which the next iteration of the **for** loop would begin. Press the **<F7>** key, and you will see the program output to this point. Your screen will appear as follows:

```
   4 1.310796 1.360489 1.464100
  5   1.402551 1.469328 1.610510
  6   1.500730 1.586874 1.771560
  7   1.605781 1.713824 1.948716
  8   1.718186 1.850930 2.143588
  9   1.838458 1.999004 2.357946
 10   1.967150 2.158924 2.593741

C> csd infl
C Source Language Debugger Version 1.1
Copyright 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading source..
Loading...

C> csd infl
C Source Language Debugger Version 1.1
Copyright 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading source...
Loading...
 1    1.070000 1.080000 1.100000

Reloading...
 1    1.070000 1.080000 1.100000
```

Once you single-stepped to the end of the **for** loop, **infl** generated one line of output, which you can see at the bottom of the screen.

If you wish, you can single-step through portions of a program. For example, you can locate the area in which a program goes astray by setting tracepoints; then, you can single-step through that area to find the exact line on which the problem occurs. Press **<F8>** to return to the source window.

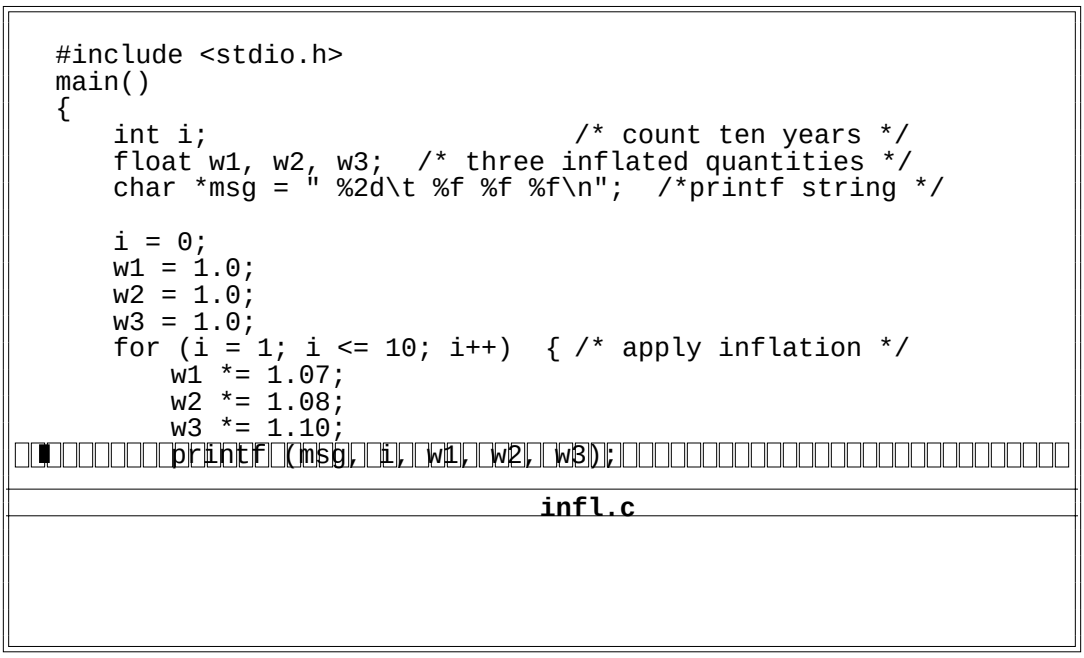
Displaying variables

csd lets you type expressions from your program in order to display the value of variables. You can evaluate any legal C expression, even function calls.

Local variables are declared at the beginning of a function. The local variable you wish to evaluate must be defined within the current scope. The *scope* of a local variable is the part of the program between the braces '{ }' of the function in which that variable is declared. Global, or *external*, variables are available from within any function's scope because they are defined outside of all functions.

Before entering local variables into **csd**'s evaluation window, you must first position the cursor in the source window so that it is between the braces of the function in which the variable is defined. Then switch to the evaluation window, and type in the variable you wish to see.

To illustrate how to display the value of variables and expressions, run the program to the end by typing **<F4>** followed by <End> . Now, position the cursor at the **printf** statement and type **<F3>** to set a tracepoint. Type **<F4>**, then **<F3>**: this tells **csd** to execute until it reaches the tracepoint. The screen now appears as follows:



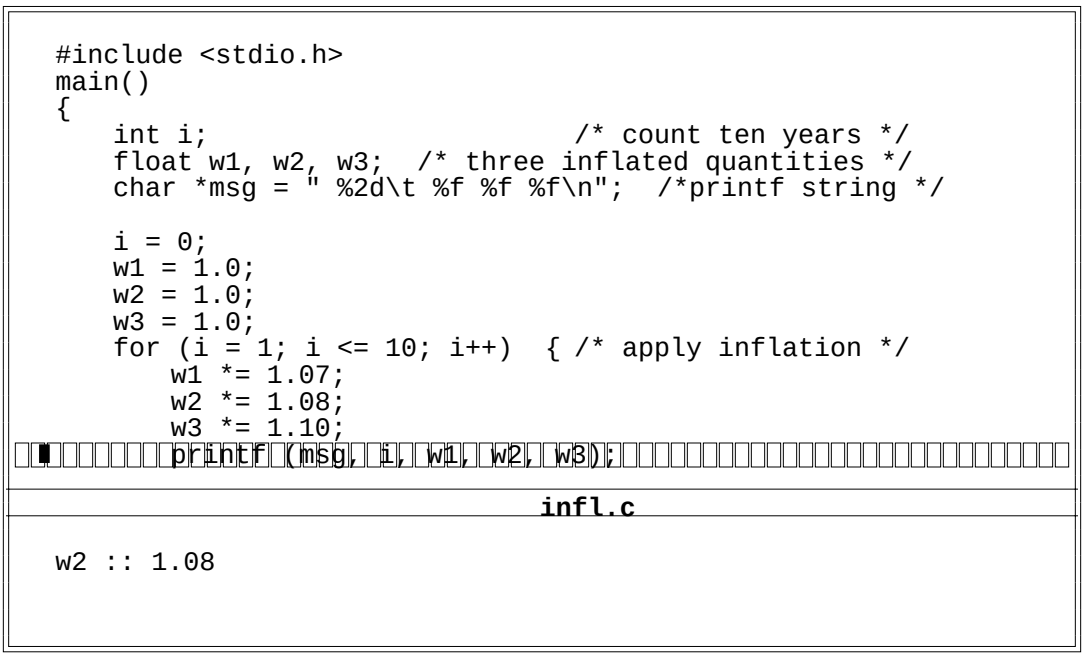
Now, switch to the evaluation window by pressing the **<F9>** key. The cursor is now flashing at the line below the reverse video line that says **infl.c**; this area of the screen is the evaluation window.

To examine the value of variable **w2**, type:

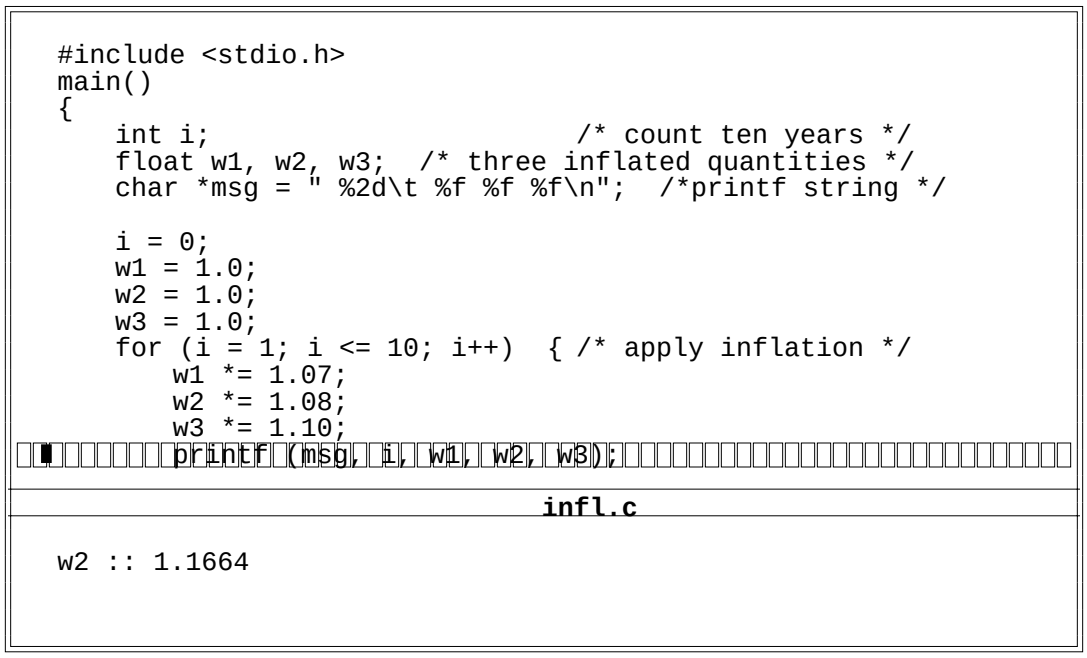
```
w2
```

If you make a mistake typing in the evaluation window, use the backspace key to delete the error. You can delete an entire line by pressing **<ctrl-U>** as long as you have not yet typed **<return>**. When you have deleted the mistake, retype the expression.

As soon as you type **w2** followed by **<return>**, the screen will appear as follows:



Notice that the value of **w2** is written after two colons, '::'. Typing **<F4>** and **<F3>** causes the program to cycle through the **for** loop once more; as you can see, the value of **w2** changes to reflect this further execution of the program. The screen now appears as follows:



Remember that all evaluation expressions in the current scope are re-evaluated in the evaluation window whenever the debugger stops execution: whether for a traced statement, a traced expression, the end of the program, or for single-stepped execution.

You can evaluate all legal C expressions, even those that you did not anticipate when you wrote the program. To see how this works, return to the evaluation window (if your cursor is not there) by typing **<F9>**. Now, enter the following expression by typing:

```
w1 + w3
```

The screen will show:

```
#include <stdio.h>
main()
{
    int i;                          /* count ten years */
    float w1, w2, w3;  /* three inflated quantities */
    char *msg = " %2d\t %f %f %f\n";  /*printf string */

    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++)  { /* apply inflation */
        w1 *= 1.07;
        w2 *= 1.08;
        w3 *= 1.10;
        printf(msg, i, w1, w2, w3);
```

infl.c

```
w2 :: 1.1664
w1 + w3 :: 2.3549
```

csd displays the results of adding **w1** with **w3** without your having to edit your program and recompile.

You can also show the value of a string pointer as a string, rather than as the address of the first character of the string, which is the way your program understands it. To do so, use **(str)** to *cast* the variable to a string type. For example, to see the value of the variable **msg** in both its forms, type the following:

```
msg
(str)msg
```

The screen will show:

```
#include <stdio.h>
main()
{
    int i;                          /* count ten years */
    float w1, w2, w3;  /* three inflated quantities */
    char *msg = " %2d\t %f %f %f\n";  /*printf string */

    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++)  { /* apply inflation */
        w1 *= 1.07;
        w2 *= 1.08;
        w3 *= 1.10;
        printf(msg, i, w1, w2, w3);
```

infl.c

```
w1 + w3 :: 2.3549
msg :: 0x0124
(str)msg :: " %2d\t %f %f %f\n"
```

The value shown for **msg**, which may be different on your screen, is the address of the first character of its string: that of **(str)msg** is the string itself.

Note that if you enter an invalid variable in the evaluation window, **csd** prints an error message; you will not be able to continue debugging until the invalid variable is removed.

Getting help

Now that you have walked through a simple program and tried **csd**'s basic debugging features, try **csd** on a program of your own. At any time while you are using **csd**, you can get on-line help with its functions by typing the *help* key, **<F6>**.

Try typing **<F6>**. **csd** displays some general help information. Now, type any of the keys listed at the bottom of the general help screen. **csd** displays help information for the function you selected. For example, while the general help screen is displayed, pressing **<F3>** will show the help screen that tells you how to trace statements and expressions.

If **csd** gives you the error message

helpfile.**hlp**: cannot open

instead of a help screen, it means that **csd** cannot find the help files in the current *path*. Use the -H option to tell **csd** where to find its help files. See section 6, *Commands reference*, for complete information about setting paths for **csd**.

When you are finished with the help feature, type **<F6>** again. The help menu will disappear, and the source window will be restored.

Finally, type **<Shift-F1>** to exit **csd** and return to .

Where to go from here

This section demonstrated the basics of **csd** on a simple program. It showed how to load a program and display its source code and output. It also discussed how to execute a program a line at a time or up to the next tracepoint, and how to examin the value of variables by typing an expression into

the evaluation window.

The next section, *Advanced features*, will expand on the uses of these features. It will also demonstrate **csd**'s more powerful capabilities, such as its ability to trace expressions and function calls within the evaluation window.

Advanced features

The previous section of this manual demonstrated the basics of **csd**, including how to set tracepoints and evaluate variables. This section describes the advanced uses of the same features, plus more powerful capabilities, such as tracing expressions and function calls from within the evaluation window.

Compiling for debugging

The examples in this section use a program called **factor**. This program, which is included on your **csd** distribution disk, calculates the prime factors of an integer. **factor** consists of two source modules, **factor.c** and **atod.c**, and was compiled for use with **csd** by using the command:

```
cc -f -VCSD factor.c atod.c -lm
```

When **csd** displays source code built from more than one source module, it displays the modules in the order in which they were entered with the compile command. Every source module that you wish to debug must be compiled with the **Let's C** compiler, using the **-VCSD** option to the **cc** command. If you are linking existing object modules into a **.exe** file, you must also use the **-VCSD** option: this tells the linker to include special object files from **libc** that are necessary for **csd**.

You may link in objects that are not compiled with **-VCSD**. In the evaluation window, the global variables of these objects may be used just as the variables of your source, but **csd** will have no knowledge of the internal variables of objects not compiled with **-VCSD**.

The **-lm** at the end of the command line tells **cc** to link the mathematics library into this program.

You can also use **csd** to debug large model programs. To invoke the large model debugger, type

```
lcsd <filename>
```

You can find more information on compiling for debugging and large and small model **csd** in section 6, *Commands reference*.

The source code for the sample program, **factor**, is included on your **csd** distribution disk.

The source code for the module **factor.c** is as follows:

```
/*
 * Factor prints out the prime factorization of numbers.
 * If there are arguments, it factors them.
 * If there are no arguments, it reads stdin until
 * either EOF or the number zero or a non-numeric
 * non-white-space character.  Since factor does all of
 * its calculations in double format, the largest number
 * which can be handled is quite large.
 */

#include <stdio.h>
#include <math.h>
#include <ctype.h>

#define NUL     '\0'
#define ERROR   0x10    /* largest input base */
#define MAXNUM  200  /*max number of chars in number */
```

```
main(argc, argv)
int            argc;
register char  *argv[];
{
       register char *chp;
       double n;
       double atod();
       char *getnum();

       if (argc != 1)
              while ((chp=*++argv) != NULL &&
                     (n=atod(chp)) != 0)
                     factor(n);

       else
              while ((chp=getnum()) != NULL &&
                     (n=atod(chp)) != 0)
                     factor(n);
       return (0);
}

/*
 * Print a fatal error message.
 */
die(str)
char *str;
{
       fprintf(stderr, "%r\n", &str);
       exit(1);
}
usage()
{
       die("Usage: factor [number number ...]");
}

/*
 * Input a number from the keyboard.
 */
char *
getnum()
{
       register char *chp,
                     ch;
       static char res[MAXNUM+1];
```

```
        do {
                ch = getchar();
        } while (isascii(ch) && isspace(ch));
        if (!isascii(ch) || todigit(ch) == ERROR)
                return (NULL);
        for (chp=res; isascii(ch) && !isspace(ch);
                ch=getchar())
        if (chp < &res[MAXNUM])
                        *chp++ = ch;
        if (chp >= &res[MAXNUM])
                die("number too big");
        *chp++ = NUL;
        return (res);
}

/*
 * Factor is the routine that actually
 * factors the double 'n'.
 * It writes the prime factors to standard output.
 */
factor(n)
double  n;
{
        double          temp,
                        limit,
                        try;
        while (n > 1 && modf(n/2, &temp) == 0) {
                printf("2 ");
                n = temp;
        }
        limit = sqrt(n);
        for (try=3; try <= limit; try += 2) {
                if (modf(n/try, &temp) != 0)
                        continue;
                do {
                        printf("%.0f ", try);
                        n = temp;
                } while (modf(n/try, &temp) == 0);
                limit = sqrt(n);
        }
        if (n > 1)
                printf("%.0f", n);
        putchar('\n');
}
```

The source code of **atod.c** is as follows:

```
#include <ctype.h>
#define ERROR   0x10            /* largest input base */
```

```
/*
 * atod() converts the string 'num' to a double and returns
 * its value.  If there is a non-digit in the string,
 * or if there is an overflow, then atod() exits with an
 * appropriate error message.  atod() accepts leading zero
 * for octal and leading 0x for hexidecimal; in the latter
 * case, 'a'-'f' and 'A'-'F' are accepted as digits.
 */
double
atod(num)
char    *num;
{
        register char  *str;
        register int   i;
        double         res     = 0,
                       base    = 10;

        str = num;
        i = *str++;
        if (i == '0')
                if ((i = *str++) == 'x') {
                        i = *str++;
                        base = 0x10;
                } else
                        base = 010;
        for (; i != ' '; i = *str++) {
                i = todigit(i);
                if (i >= base)
                        die("bad number '%s'", num);
                res = res * base + i;
                if (res+1 == res)
                        die("number too big '%s'", num);
        }
        return (res);
}

/*
 * todigit() converts character 'ch' to an integer equivalent,
 * assuming that 'ch' is a digit or 'a'-'f' or 'A'-'F'.
 * If this is not true, then it returns ERROR.
 */
todigit(ch)
register int   ch;
{
        if (!isascii(ch))
                return (ERROR);
        if (isdigit(ch))
                return (ch - '0' + 0);
        if (isupper(ch))
                ch = tolower(ch);
        if ('a' <= ch && ch <= 'f')
                return (ch - 'a' + 0xA);
        return (ERROR);
}
```

Running without csd

csd C source debugger

When you compile a program with the **-VCSD** option, you can run it just like any other program. To illustrate how **factor** works, type the following line:

```
factor 12
```

After a moment, it prints the result:

```
2 2 3
```

This result means that the number 12 is produced by multiplying the prime numbers two, two, and three.

Evaluation window

As described in the previous section, you can ask **csd** to print the values of variables or expressions by typing them into the *evaluation window*. An evaluated expression may *not* use keywords (such as **if** or **for**), labels, braces '{' '}', or semi-colons ';'. Additionally, **csd** reserves the types **(oct)**, **(hex)**, and **(str)** for use as casts to display values in base 8, base 16, and as character strings, respectively. You may redeclare these identifiers locally or globally, but you will lose the ability to cast expressions to hexadecimal, octal, or character array display.

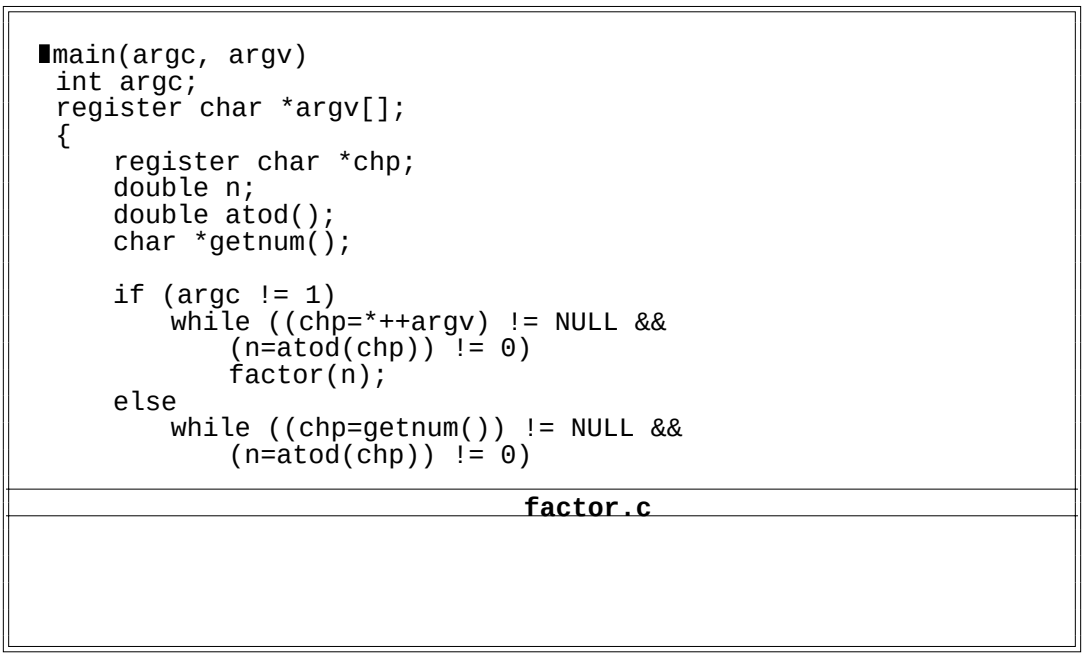
csd evaluates the expressions you enter in the evaluation window when you first type them in and press **<return>**. It also evaluates the expressions you have typed in the window whenever the program stops executing--that is, when it reaches the end of the program, reaches a tracepoint, or when you single-step through the program.

To illustrate the use of the evaluation window in detail, invoke **csd** with the sample program **factor** by typing:

```
csd factor 234
```

or by using the Mark Williams shell, MWS, as described above. In a moment, **csd** will draw the source window on the screen, where you work interactively with **csd**.

Now press the *find* key, **<F1>**, to invoke **csd**'s *find* command. When the prompt appears, type **main**. **csd** will find the beginning of the **main** function and move the cursor there. The screen will appear as follows:



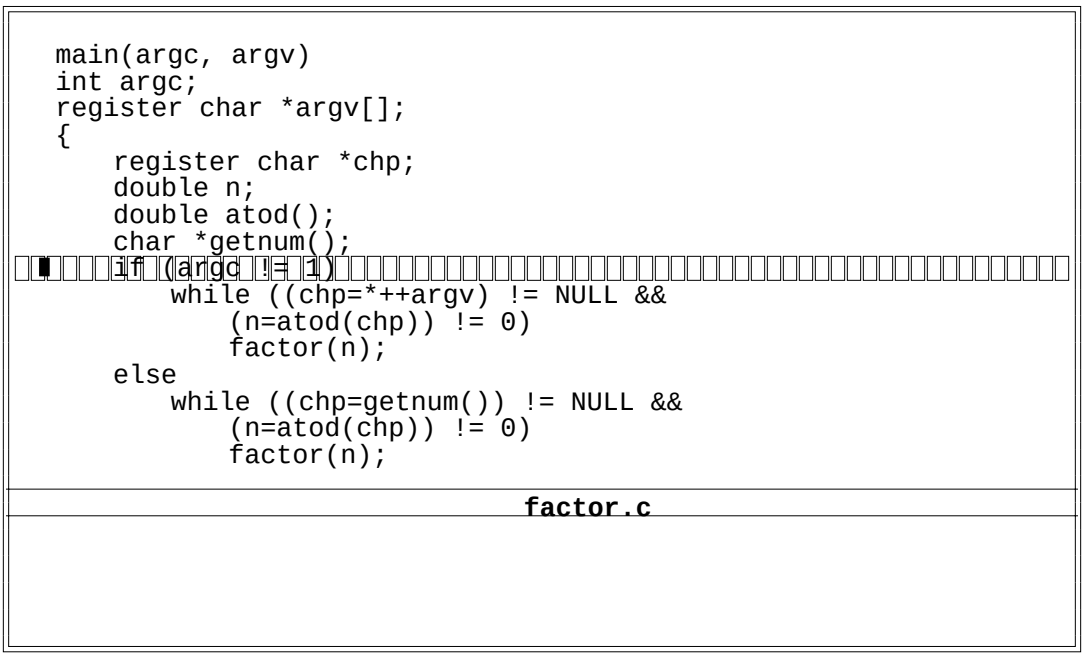
Press the cursor-movement key **<↓>** or **<ctrl-N>**, until the cursor is at the line

```
if (argc != 1)
```

Now, set a tracepoint on that line by typing **<F3>**. Remember that tracepoints can be set only on executable statements. If you try to set a tracepoint on a non-executable statement, **csd** gives you the error message:

```
not executable statement
```

The screen now appears as follows:



To run the program to this tracepoint, type **<F4>.** The following prompt will appear at the bottom of the screen:

```
Run:  Trace F3 Return D <Home>  <End>  <F5>  <F6>
```

This tells you that you can run a program to a number of different points within the program. For example, Run followed by Trace tells **csd** to run to the next tracepoint. Run followed by return is the command to step through the program a line at a time. A complete description of each of these keys can be found in section 6, *Commands reference*. You can find the keys for each of these points on the bank of function keys or on the numeric keypad.

For the current example, type **<F3>**. **csd** will now execute your program from its beginning until it reaches the tracepoint you set a moment ago. **csd** briefly displays the *program window* (which is where your program prints its normal output); then it restores the source window with the cursor positioned at the line causing the trace stop.

Now, press **<F9>**. This moves the cursor into the *evaluation window*, which is where you can type an expression for **csd** to evaluate. **csd** lets you evaluate any variable that is either global or local to the function within which the cursor was positioned in the *source window*. In this instance, you can now examine any variable that is global or local to **main**. Remember that if you enter an invalid variable name or an incorrect expression into the evaluation window, you must erase the characters using the backspace key, or **<ctrl-U>** to erase the entire line.

To see how you can evaluate a variable in the evaluation window, type:

```
argc
```

csd will respond by printing the value of **argc** in the evaluation window. The screen now appears as follows:

```
main(argc, argv)
int argc;
register char *argv[];
{
    register char *chp;
    double n;
    double atod();
    char *getnum();
    if (argc != 1)
        while ((chp=*++argv) != NULL &&
            (n=atod(chp)) != 0)
        factor(n);
    else
        while ((chp=getnum()) != NULL &&
            (n=atod(chp)) != 0)
            factor(n);
```

factor.c

```
argc :: 2
```

argc is the number of arguments passed to a program. In this instance, the program **factor** has two arguments: the name **factor** itself and the number to be factored, **234**.

While you are still in the evaluation window, type:

```
argv[1]
```

csd will evaluate this variable, and the screen will show:

```
main(argc, argv)
int argc;
register char *argv[];
{
    register char *chp;
    double n;
    double atod();
    char *getnum();
    if (argc != 1)
        while ((chp=*++argv) != NULL &&
            (n=atod(chp)) != 0)
            factor(n);
    else
        while ((chp=getnum()) != NULL &&
            (n=atod(chp)) != 0)
            factor(n);
```

factor.c

```
argc :: 2
argv[1] :: 0x5333
```

Note that pointers are automatically displayed in hexadecimal. The number that appears in the evaluation window after **argv[1]** may be different on your system, depending on how memory is allocated. **argv** points to the array of arguments passed to **factor**. These arguments are usually pointers to character strings.

To see the string that the pointer points to, use the **csd** cast **(str)**. Type:

```
(str)argv[1]
```

The screen now shows the following:

```
main(argc, argv)
int argc;
register char *argv[];
{
    register char *chp;
    double n;
    double atod();
    char *getnum();
    if (argc != 1)
        while ((chp=*++argv) != NULL &&
            (n=atod(chp)) != 0)
            factor(n);
    else
        while ((chp=getnum()) != NULL &&
            (n=atod(chp)) != 0)
            factor(n);
```

factor.c

```
argc :: 2
argv[1] :: 0x5333
(str)argv[1] :: "234"
```

The string “234” is from the command line

```
csd factor 234
```

You can enter any valid C expression into the evaluation window as long as the names in the expression are either global or local to the current function. This includes the ability to call functions. Type

```
printf("Here’s a csd string.\n")
```

csd’s evaluation window now appears as follows:

```
argv[1] :: 0x5333
(str)argv[1] :: "234"
printf("Here’s a csd string.\n") :: 0
```

csd shows its value as ‘0’. To see the result of the **printf** command that you entered in the evaluation window, press the **<F7>** key. **csd** will jump back to the program window, where your program prints its normal output. The program window appears as follows:

```
C>factor 12
2 2 3

C>csd factor 234
C Source Language Debugger Version 1.1
Copyright (c) 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading source...
Loading...
Here's a csd string.
```

csd executed the **printf** function as if it were part of the program **factor**, and wrote the string **Here's a csd string** onto the program window.

Using the select key

Now, return to **csd**'s source window by typing **<F8>**.

Note that the first expression you had typed into the evaluation window is no longer visible. This is because the evaluation window can display only a fixed number of expressions at once. To display more expressions in the evaluation window, you can increase the size of the evaluation window with the *select* key. Typing **<F2>** followed by the <↑> or <↓> key raises or lowers the boundary between the evaluation and source windows, respectively. You can also use the select key to choose which screen is displayed while your program is executing. For example, if you want to watch traced statements displayed on the history window, type **<F2>** followed by **<F10>**. You can also choose to display the source window during the execution of your program. Type: **<F2>** followed by **<F8>**.

The following exercise demonstrates this feature. First, exit from **csd** by typing **<Shift-F1>**. This returns you to MS-DOS. Now, invoke **csd** again by typing:

```
csd factor 234
```

When the source window appears, move the cursor to the open brace '{' that follows **main**; then jump into the evaluation window by typing **<F9>**. Type the expression

```
argc
```

but follow it with **<F3>** rather than **<return>**. **argc** will be highlighted in the evaluation window to show that **csd** will trace the variable **argc**. When you trace an expression in the evaluation window, **csd** evaluates the expression *before* a line of code is executed. As it traces **argc**, **csd** will run slowly and you can view the program's execution.

Now, return to the source window by typing **<F8>**. To see the execution proceed, type **<F2>** followed by **<F8>**. With these keys, you tell **csd** to display the source window while the program is executing. Start execution by typing **<F4>** then <End> . As you can see, the source window scrolls through the program as it executes. The bottom line of the evaluation window displays the message

```
program running
```

The value of **argc** is displayed in the top line of the evaluation window. Because you used the **<F2>** key to display the source window during the execution of the program, no output is written onto the program window. Instead, any output appears in the selected window. It can be removed by switching screens. This output, unlike that in the program window, is lost once you switch windows.

Displaying the source window during program execution allows you to see both the program window and the evaluation window at the same time. Using this technique lets you watch the values of variables and expressions change in the evaluation window while your program executes.

Exploring the stack

When you evaluate variables and expressions within the evaluation window, you can examine only global variables or variables that are local to the current function (that is, the function within which the cursor is positioned in the source window). However, **csd** also lets you examine and modify variables that are local to the function that called the current function.

When one C function calls another function, the variables local to the *calling* function are stored in a special area of memory called the *stack*. By examining the stack, you can read the values of the calling function's variables; by editing the stack, you can change the values of these variables. The *In* key, <→> and the *Out* key, <←> let you work with the stack.

The following exercise demonstrates the <→> and <←> keys. First, exit from **csd** by typing **<Shift-F1>**. Then, restart **csd** with the program **factor** by typing:

```
csd factor 234
```

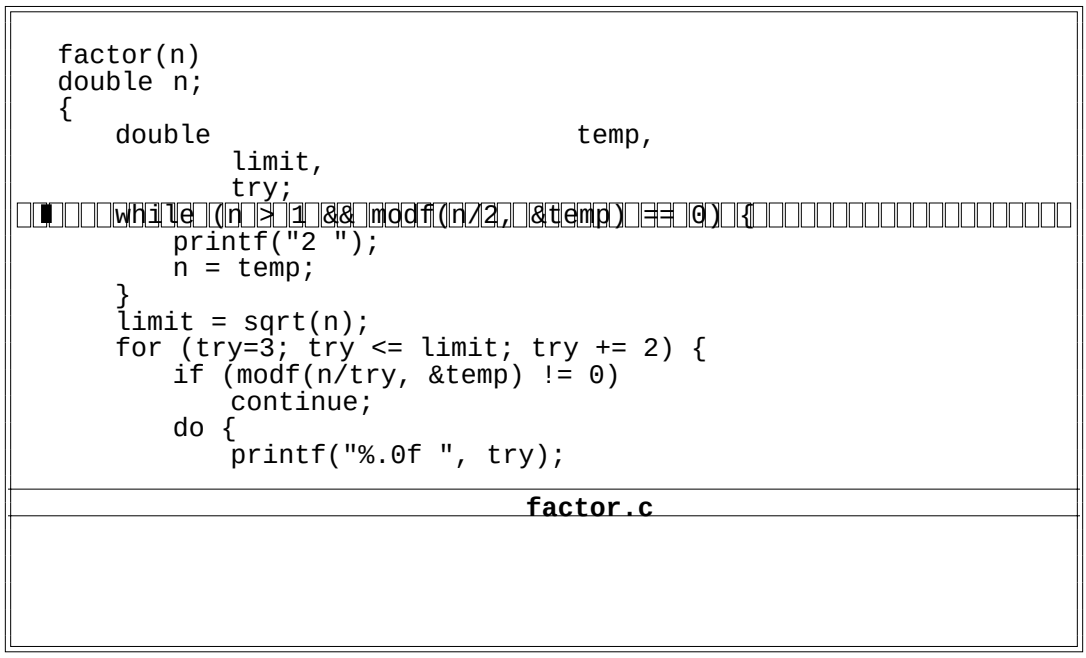
You can easily find the *function* **factor** within the program. Type **<F1>** followed by

```
^factor
```

That is, type a caret '^' plus the string **factor**; then type **<return>** or **<↓>**.

The string **^factor** is a *pattern*, or *regular expression*, that **csd** seeks. The caret '^' tells **csd** to look for **factor** only at the beginning of a line. The caret is one of the special characters that you can use with **<F1>** to search for specific patterns. A list of these characters and their uses can be found in the *Commands reference*, later in this manual.

The cursor is now positioned within the function **factor**. Press <End> to move the cursor to the line that begins with **while**. Type **<F3>** to set a tracepoint on this line. Your screen now appears as follows:



Now, type **<F4>** followed by **<F3>** to run the program up to the tracepoint.

Once this is done, the program is poised to execute the **while** statement within the function **factor**. The scope, or *stack frame*, that is currently active is that of the function **factor**, and you can now evaluate variables which are local to it.

You can determine what function called **factor** and activate the stack frame of the calling function by typing <←> . Try it. The screen now shows:

```
double     n;
double     atod();
char       *getnum();

if (argc != 1)
    while ((chp=*++argv) != NULL &&
        (n=atod(chp)) != 0)
    factor(n);
else
    while ((chp=getnum()) != NULL &&
        (n=atod(chp)) != 0)
        factor(n);
return (0);
}
die(str)
char   *str;
```

factor.c

The cursor has jumped to the line within **main** that called **factor**. The parameters of **main** are now accessible.

Pressing the <→> key will return you to the **while** statement in **factor**.

With the <←> key you can explore all enclosing stack frames. For example, if a program had a function that called another function that called yet a third function, you could position the **csd** cursor within the last-called function and then use the <←> key to check the values of its two "ancestors". The present example, however, has only two such frames: **main** and the local level.

Changing variables

Comma expressions (expressions in which the comma is used as an operator) and assignment statements are among the legal C expressions that you can enter into the evaluation window. Expressions with side effects (for example, expressions involving operators) will propagate these effects whenever they are evaluated. *Before you proceed, please note that you should not change the variable in the following example: doing so causes the program to run indefinitely.* For example, if you were to type

```
n=12
```

you would change the value of **n**. If you then tried to execute this program to the end, it would never terminate because the assignment statement **n=12** would be executed each time a tracepoint was encountered.

You should remove assignment statements and other statements with side effects from the evaluation window as soon as you are finished with them.

Now press **<Shift-F1>** to exit from **csd**.

Where to go from here

This section described advanced **csd** techniques. You can now explore your source with the *Find* key, **<F1>**, and search for lines that contain specific character patterns. Advanced evaluation window techniques were shown, and a method of viewing the execution of your program in real time

was demonstrated. With the <→> and <←> keys, you can explore the stack. You can also change the value of variables by placing assignment statements in the evaluation window, although this must be done with care.

The following section, which concludes this manual's tutorial, presents a sample debugging session. Section 5, *Questions and Answers*, presents commonly asked questions about **csd**, and their answers. If you are having any difficulty with **csd**, check here first.

Section 6, *Commands reference*, summarizes **csd** commands. It describes the meaning of each command key and all combinations of command keys. It also suggests some uses for **csd** commands that are beyond the scope of a tutorial.

A sample debugging session

This section of the manual contains exercises that let you use **csd** on a program with bugs. An error-filled version of the program **infl** named **inflbug** will be used to demonstrate some common C language bugs and how to find them with **csd**. **inflbug.c** is included on your **csd** distribution disk.

To begin, compile **inflbug.c** using the following command line:

```
cc -f -A -VCSD inflbug.c
```

Note the **-VCSD** option, which tells the compiler to include the information needed to run your program under **csd**. Note also the **-A** option. This option automatically calls the MicroEMACS screen editor when the compiler encounters an error during compilation. The errors are displayed in one window, and the source code file in the other, with the cursor set to the line number indicated by the first error message. Typing **<ctrl-X>>** moves to the next error, and **<ctrl-X><** moves to the previous error. To recompile, close the edited file with **<ctrl-Z>**. Compilation will continue either until the program compiles without an error, or until you exit the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

inflbug.c will compile without an error message.

Where to start

As you know from section 2, the program **infl** computes three different rates of inflation over a span of ten years. **inflbug** is supposed to do exactly that, but fails. To set up this example, the errors in **inflbug.c** have been created so that the program will compile but not execute.

Type

```
inflbug
```

at the MS-DOS prompt. **inflbug** runs, but instead of producing a chart of inflated values as the program **infl** did in section 2, it does nothing. If you had written this program, you would have planned for the program to compute the inflated values ten times, and print the results in a chart to your screen. Since the program compiled without an error message, you can now use **csd** on the program to find just where the program fails.

Type

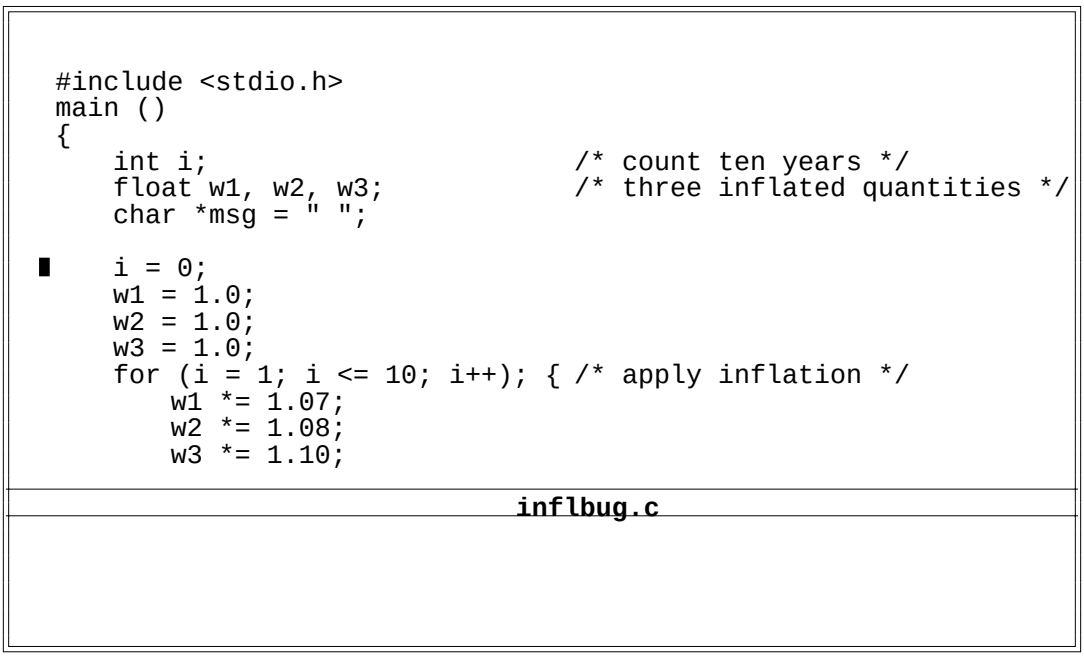
```
csd inflbug
```

The source window will appear, with the beginning of the program displayed in the source window. Your screen will appear as follows:

```
#include <stdio.h>
 main ()
 {
     int i;                          /* count ten years */
     float w1, w2, w3;  /* three inflated quantities */
     char *msg = " ";

     i = 0;
     w1 = 1.0;
     w2 = 1.0;
     w3 = 1.0;
     for (i = 1; i <= 10; i++); { /* apply inflation */
         w1 *= 1.07;
         w2 *= 1.08;
         w3 *= 1.10;
```

inflbug.c

To begin debugging, single-step through the program. Type **<F4>** followed by **<return>** to execute one line of **inflbug**. The screen will look as follows:



Notice that the cursor is positioned at the first executable line of code. Continue single-stepping through the program by typing **<F4>** followed by **<return>**. After each step, type **<F7>** to change from the source window to the program window after executing each line to see what output, if any, the program has produced. Return to the source window by typing **<F8>**. As the program executes each line, the cursor will move to that line of code.

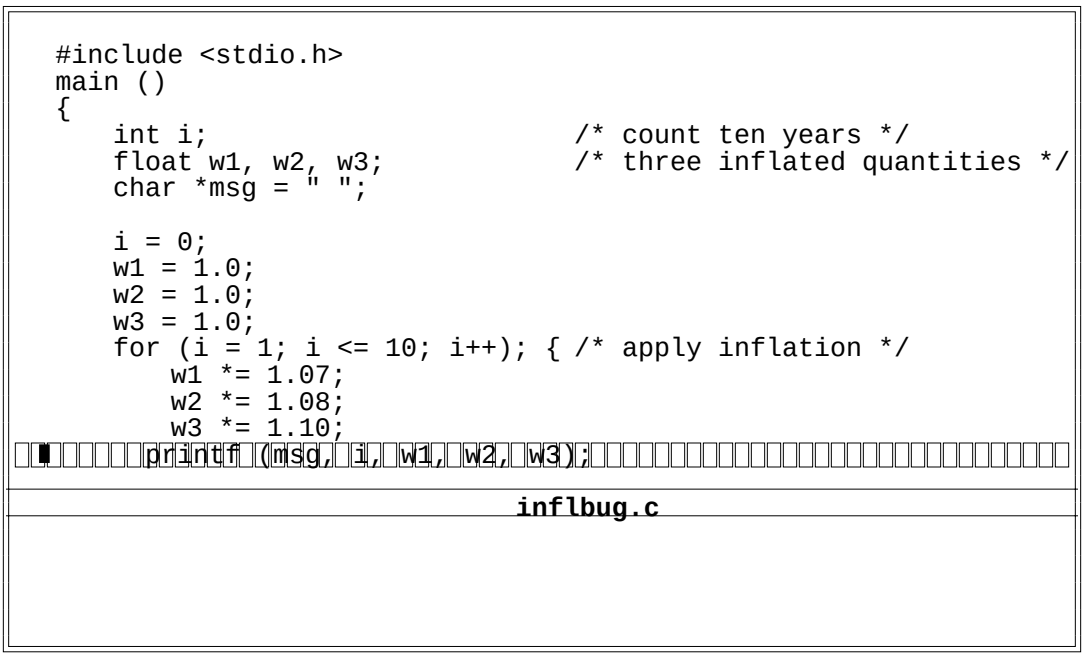
As you watch **inflbug** execute each line, you will notice that you must type **<F4>**, then return ten times after you reach the 'for' loop before the cursor moves on to the statements following the **for** statement. It then proceeds through the rest of the loop, stopping once before each statement. Then the program exits, leaving the cursor positioned at the upper left corner of the source window.

Switch to the program window by typing **<F7>**. The program window looks like this:

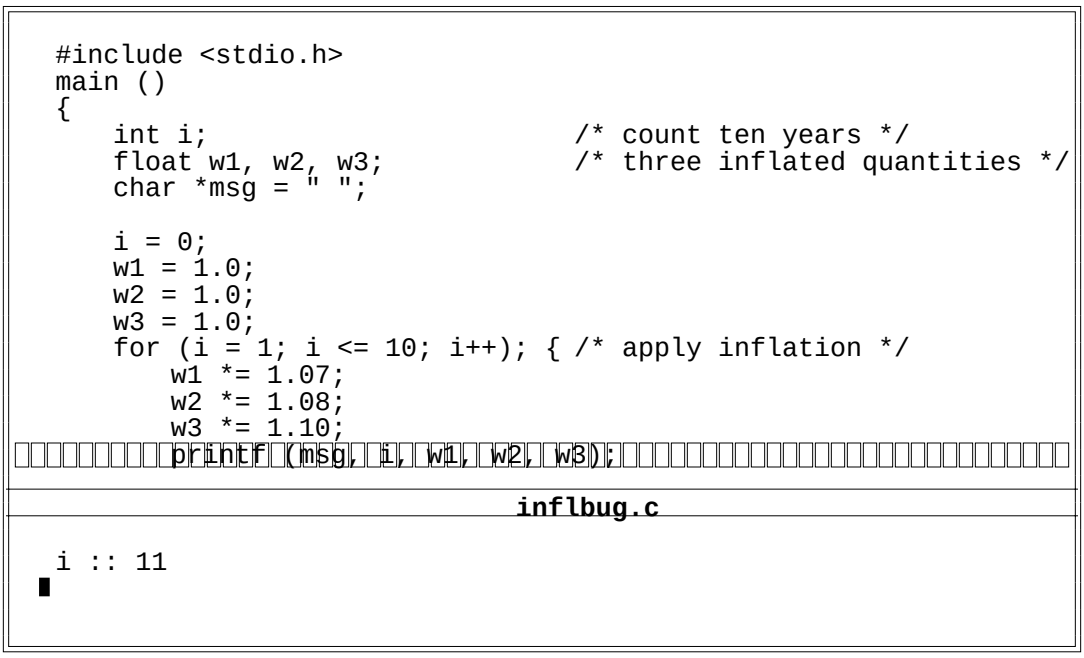
```
C>inflbug


C> csd inflbug
C Source Language Debugger Version 1.1
Copyright (c) 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading source...
Loading...
```

The program should execute the entire **for** loop; instead, it proceeds only with the **for** *statement*, incrementing the variable **i**. It does not perform the operations in the rest of the loop until after it has executed the **for** statement ten times. It seems as if the **for** statement is isolated from the rest of the loop. Type **<F8>** to return to the source window.

To isolate the problem further, set a tracepoint on the **printf** statement. Use the **<↓>** key or **<crtl-N>** to position the cursor at the **printf** statement, then type **<F3>**. The **printf** function is the last executable statement of the **for** loop. By stopping program execution here, you can see the value of the program's variables after one pass through the loop. Now, run the program to the traced point by typing **<F4>** then **<F3>.** This causes the debugger to execute the program for what should be one pass through the **for** loop. Your screen looks like this:



Check the value of **i**: move to the evaluation window by typing **<F9>**. Type **i** and **<return>**. The line in the evaluation window shows



i is a variable that is incremented to set a test-condition for execution of the **for** loop. Each time the loop is executed, the value of **i** increases by one. Since the variable has a value of 11, you know that the **for** statement has been executed ten times before it reached the traced **printf** at the end of the loop. Now, check the value of **w1** to see if it has been affected by the statement

```
w1*=1.07;
```

The operator '*=' should multiply the variable **w1** (which has a declared value of one), by 1.07, then assign the product as the new value for **w1**. Type

```
w1
```

in the evaluation window, then **<return>**. The results will be:

```
#include <stdio.h>
main ()
{
    int i;                          /* count ten years */
    float w1, w2, w3;               /* three inflated quantities */
    char *msg = " ";

    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++); { /* apply inflation */
        w1 *= 1.07;
        w2 *= 1.08;
        w3 *= 1.10;
        printf (msg, i, w1, w2, w3);
```

inflbug.c

```
i :: 11
w1 :: 1.07
```

This means that the variable **w1** has been multiplied by the inflation rate once, even though the **for** statement has made ten iterations. From this information, you can conclude that even though the **for** loop has been incrementing, the steps within the loop have not been executed. Somehow the test-condition statement is separated from the rest of the instructions in the loop.

Take a closer look at the **for** statement:

```
for (i = 1; i <=10; i++);
```

The loop test says that while **i** is less than or equal to 10, increment it by one; but it ends there, and does not continue with the rest of the loop because of the semi-colon ';' at the end of the line. So, **csd** runs the **for** statement instructions the designated number of times and goes on to the statements which apply the inflation rate. It proceeds to the **printf** statement, then exits.

Editing your program

Once you have tracked down the bug in this program, you need to edit out the extra semi-colon and recompile the program. Exit from **csd** by typing **<Shift-F1>**. Use MicroEMACS to edit your program. The source code for **inflbug** is in the file **inflbug.c**. To open the file, type

```
me inflbug.c
```

The screen clears, and in a moment the source code appears. Type **<ctrl-N>** to move the cursor to the **for** statement, then type **<ctrl-F>** until the cursor is positioned just to the right of the extra semi-colon at the end of the **for** statement. Type the delete key **<Del>** to remove the semi-colon. Now save the text and exit MicroEMACS by typing **<ctrl-Z>**. For more information on how to use the editor, see the MicroEMACS tutorial in the manual for **Let's C**.

Recompiling your program

If you change your source file, as you have in this tutorial, you must recompile the program with the **-VCSD** option on the compile command line. After you have returned to the prompt, type this compile command line:

```
cc -f -A -VCSD inflbug.c
```

As discussed at the beginning of this section, the **-A** option on the command line tells the compiler program to invoke MicroEMACS whenever it encounters an error message. If you have made any errors while editing the source code text, you will be returned to MicroEMACS to correct the program. Exiting from MicroEMACS automatically recompiles the program. This compile-edit-compile cycle will continue until your program is error free, or until you exit by typing **<ctrl-U>**, followed by **<ctrl-X><ctrl-C>**.

When you have finished compiling, try running your program again. At the system prompt, type

```
inflbug
```

Another bug

The program still produces no output. Invoke **csd** on the program again. Type

```
csd inflbug
```

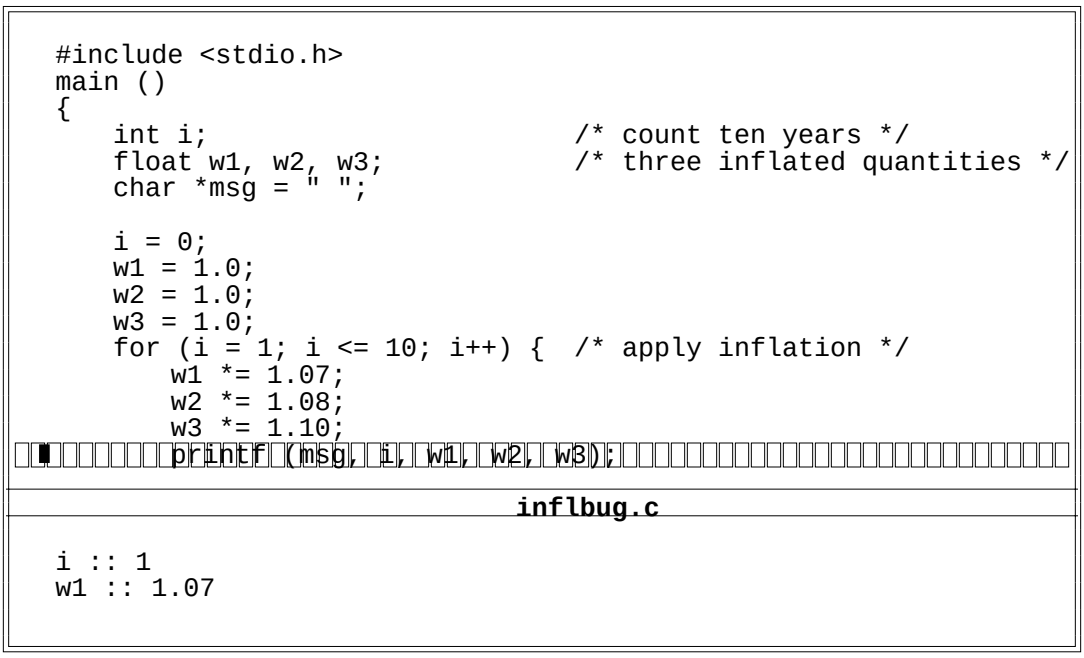
and **<return>**.

Using the cursor movement keys, position the cursor at the **printf** call, and type **<F3>** to set a trace point.

Press **<F4>** followed by **<F3>**, and the program will execute to the traced line. Now, enter the evaluation window by typing **<F9>**. Type

```
i
w1
```

csd will return the following values in the evaluation window:



Now run the program through again by typing **<F4>** followed by **<F3>**. The program will run once through the loop. Type **<F4>** then **<F3>** repeatedly, and watch the value of the variables you typed in the evaluation window change. When you have run **inflbug** to its end, your screen will appear as follows:

```
#include <stdio.h>
main ()
{
    int i;                          /* count ten years */
    float w1, w2, w3;               /* three inflated quantities */
    char *msg = " ";

    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++) {  /* apply inflation */
        w1 *= 1.07;
        w2 *= 1.08;
        w3 *= 1.10;
        printf (msg, i, w1, w2, w3);
```

inflbug.c

```
i :: 10
w1 :: 1.967150
```

By checking the value of **i** and **w1** with each iteration of the **for** loop, you can see that the program is operating upon the variables and executing the loop properly. The variable **i** has been incremented for the test-condition of the **for** loop, and the variable **w1** has been operated on with the appropriate results. However, none of the computed values are being printed to the program window; therefore, the problem must be in the **printf** call.

Since **printf** is a format conversion function, it should contain instructions to format a line for printing the variables. This format has conversion specifications which take the value of **i** as the first number on the output line (the line number). The inflated variables are the rest of its arguments. **msg** is the pointer to a string which should contain the conversion specifications. Evaluate the string **msg** using the string cast **(str)**. Move the cursor so that it is inside **main**. Return to the evaluation window and type:

```
(str)msg
```

in the evaluation window. The result is:

```
(str)msg :: ""
```

The pointer has not been initialized to hold the **printf** conversion specifications, so it picks up a random value in memory. The pointer is intended to contain the address of a string, but no address has been assigned to it. The string therefore contains whatever was at that location on the stack when the program was run. **msg** does not point to a valid format, and **printf** does nothing.

Calling functions in the evaluation window

Calling functions from the evaluation window is a powerful debugging tool. In this sample debugging session, you can use this tool to test a modified **printf** call.

Type the following line in the evaluation window:

```
printf("  %2d         %f %f %f\n", i, w1, w2, w3)
```

Now, restart the program by typing **<F4>** followed by <Home> . Run the program to its end: type **<F4>** then <End> . Check the program window; you will see the effect of calling this routine in the

evaluation window. All of the **for** loop operations have been displayed in the format called for in the new **printf** statement; in addition, the **printf** statement already in the program prints its output with each iteration of the loop.

By calling a new **printf** routine, you can see that while your variables are being incremented as you planned, there is something wrong with the **printf** statement: it does not have a format string.

Exit, edit, and recompile

Exit **csd** by typing **<Shift-F1>**. Then invoke MicroEMACS on the file **inflbug.c**. Type

```
me inflbug.c
```

With **<ctrl-N>**, move the cursor to the line

```
char *msg = " ";
```

Type **<ctrl-K>** to remove that line. The cursor will be positioned at the beginning of the line. Type a **<Tab>**, then:

```
char *msg = " %2d       %f %f %f\n";
```

Now save your changes and exit MicroEMACS by typing **<ctrl-Z>**. Recompile as before, using the following command line:

```
cc -f -A -VCSD inflbug.c
```

If you run the program **inflbug** by typing

```
inflbug
```

you should get the results you expected, as follows:

```
C>inflbug
  1   1.070000 1.080000 1.100000
  2   1.144900 1.166400 1.210000
  3   1.225043 1.259712 1.331000
  4   1.310796 1.360489 1.464100
  5   1.402551 1.469328 1.610510
  6   1.500730 1.586874 1.771560
  7   1.605781 1.713824 1.948716
  8   1.718186 1.850930 2.143588
  9   1.838458 1.999004 2.357946
 10   1.967150 2.158924 2.593741
 C>
```

Where to go from here

This section used the program **inflbug** to demonstrate the use of **csd** on a program with bugs. Special attention was paid to the use of the **-VCSD** option in the command line, as well as the **-A** option to call MicroEMACS from the compile command line. Practical examples of single-stepping, setting tracepoints, and calling functions in the evaluation window were shown.

Sections 6 through 9 will give detailed references for **csd** commands, as well as copies of the **csd** help screens, an explanation of error messages, and some commonly asked questions. You have seen the use of **csd** on some programs; with the *Commands reference* section and the **csd** on-line help screens as guides, you will be able to use **csd** to debug your own C programs.

Questions and answers

Here are some questions users frequently ask about **csd**. The Mark Williams Company welcomes other questions and comments from users.

Why does the **-VCSD** *flag make objects larger?*

The option **-VCSD** puts debug information into the object module as it compiles the C program. This additional information enlarges the object module.

Does **csd** *work on a color monitor?*

Yes. In order to run **csd** on a color monitor that is not in 80x25 character mode, use the **-G** (graphics) option on the **csd** command line.

Why can't I cast to **(str)**? I cast to **(str)**?'>=29

csd reserves types **(str)**, **(oct)**, and **(hex)** for use as casts to display strings. You have defined a variable **str** in one module that you have compiled for debugging.

Why can't I cast to **str** *or call functions in the evaluation window?* I cast to **str** *or call functions in the evaluation window?*'>=29

You did not use the **-VCSD** option to **cc** when you linked your program. Recompile or relink to solve the problem.

Why can't **csd** *find the variables and functions I ask for in the evaluation window?* **csd** *find the variables and functions I ask for in the evaluation window?*'>=29

If **csd** can find the source, yet not find variables and functions in the evaluation window, you probably have an assembly language module in your program. Use the -O (model override) flag when you invoke **csd**.

Commands reference

Every source file that you wish to debug must be compiled with the **Let's C** compiler, using the **-VCSD** option to the **cc** command.

You may link in objects that are not compiled with **-VCSD**. In the evaluation window, the global variables defined in these objects will be visible, but **csd** will have no knowledge of the internal variables of objects not compiled with **-VCSD**.

Invoking csd

To start **csd**, type **csd** followed by the name of the program you want to debug. If you invoke **csd** on a program and receive the error message:

```
out of space
```

while reading source, you need to use the large model source debugger, **lcsd**. **lcsd** is the version of the debugger that has a large data segment. To invoke the large model debugger, type **lcsd** followed by the name of the program you want to debug.

For Tandy 2000 users, type **tcsd** to invoke the debugger. For any other Tandy PC compatible, invoke the debugger with the **csd** command.

To debug the program **infl**, type:

```
csd infl
```

When you invoke the debugger, it will print in the history window:

```
C Source Language Debugger Version 1.1
Copyright 1984-1988 by Mark Williams Co., Lake Bluff, IL
Reading tables...
Adding source:  infl.c
Loading infl...
```

The line

```
Reading tables...
```

shows that **csd** is reading the debug information for the program, and

```
Loading infl...
```

means that it is loading the **.exe** program file. The line

```
Adding source:  infl.c
```

means that **csd** is adding the source file used to build the program. **Adding source:** followed by a file name appears once for each source file **csd** adds.

Typing **csd** at the prompt without arguments produces the following reminder:

```
usage: CSD -G -D -O[L][M] -Hhelppath -Ssourcepath -T file[.exe] args
```

The information following "CSD" refers to command line options, which are discussed below. **file** is the name of a **.exe** file you wish to debug. The **.exe** suffix is optional. **args** is the remainder of the command line to be supplied to your program (as if it were running without **csd**). For example, when you type

```
csd factor 234
```

the number **234** is the argument passed to the program **factor**.

The **csd** options must *precede* the file name, so that they are not confused with the arguments to your program. The **csd** options may be entered in any order.

Options

The following describes the options to the **csd** command.

-O[L][M]

User program is:

-O small model object module format
-L large model object module format
-M small model Mark Williams format

Ordinarily, **csd** determines the format of the **.exe** file you are using by the number of relocations to be done at load time (this is part of the **.exe** header).

0 relocations means MWC format
1 relocation means small OMF
>1 relocations means large OMF

If a small model program contains a **.s** file with absolute segment references (the **@** operator), these references must be relocated, causing the above rule to be violated. Note that the Mark Williams format was produced by **Let's C** in versions earlier than 4.0. It is no longer used by **Let's C**. These options allow debugging of such **.exe** files.

-D*nnn* Change *data segment* size. If you are using a version of **csd.exe** called **lcsd** that has a large data segment, the **-D** option lets you specify the size of **csd**'s workspace. *nnn* is a decimal number of kilobytes; the default is 128. This workspace holds disk buffers, user expressions, and the history. It is allocated in addition to the fixed memory used to hold your program's symbol table. If this option does not appear in the usage reminder, you are using a **csd.exe** with a small data segment; if this is the case, the data segment is 64 kilobytes, the maximum possible.

-H Set the *help path*. This option tells **csd** where to find its help files. By default, **csd** looks for help files in the current directory, then in the directories named by the environmental variables **PATH** and **LIBPATH**.

The help path must be a complete MS-DOS directory specification. It should terminate in a back slash (\), the only exception being a drive specification (e.g., **C:**).

-G The *graphics* option. Use this when running **csd** on a color system that is not in 80x25 character mode.

-I*nn* The reset *interrupt* option. **csd** ordinarily uses MS-DOS interrupt 3 to set tracepoints in your program. However, if the program you are debugging already uses interrupt 3, you may want **csd** to set tracepoints using a different interrupt. The **-I** option tells **csd** to use interrupt *nn*, where *nn* is a decimal number.

-S *source path*

-T *source path*

These options tell **csd** where to find your C source files.

When you compile a C source file with the **-VCSD** option, the file name that you type (which

may include a directory specification) is saved within the object modules and the executable file. For example, if you typed

```
cc -VCSD \source\example.c
```

the executable program would remember the name of its source file not as **example.c**, but as **\source\example.c**. This name will be displayed in the reverse-video line between the source and evaluation windows.

The **-S** option tells **csd** to look for C source files in the named *source path*. **csd** will prepend the given directory path to each source file name in the symbol table. **-S** is useful if you compiled the program using the file names with no directory specification, and you want to use the debugger from another directory without recompiling. For example, if you compiled the program with

```
cc -vcsd program.c
```

and invoked **csd** with the command

```
csd -Sutility\program
```

csd will look for the source file **utility\program.c**.

The **-T** option also tells **csd** to look for C source files in the named *source path*, and it strips the existing directory information from the saved source file names. **csd** will strip any directory path names from the source file names it finds in the symbol table, then prepend the given path before reading in the files. **-T** is useful if you compiled the program giving path names that are no longer relevant and should be disregarded. For example, if you compiled the program using the command line

```
cc -vcsd \src\program.c
```

and invoked **csd** with

```
csd -T\utility\program
```

csd will look for a source file called **\utility\program.c** and ignore the leading directory name it found in the symbol table.

This feature is helpful in situations where you would rather not rebuild an executable file or change directories. For simplicity's sake, it is suggested that you run **csd** from the same directory where you compiled the program being debugged, or specify complete pathnames for the source files given to the **cc** command.

Exiting csd

To exit the debugger once you have started debugging, press the **<Shift-F1>** key. If you are currently typing a line in the evaluation window, type **<ctrl-U>** before exiting.

Getting help

Whenever the debugger is waiting for your input, you can type the **<F6>** key and the debugger will display the master help screen. This help information describes all the major functions of the debugger and the key that invokes each function. From this screen, you can choose other help screens that describe individual functions in detail.

After you have read the help screen, type **<F6>** again to return to the debugger.

The following help screens are provided:

```
Cursor movement
General
Trace
Run
Evaluation
Find
History
Insert/Delete
Select
```

The help screens are detailed enough so that many users do not need to refer to this manual once they get started. Section 6, *Help screens*, contains a copy of every help screen.

Windows

csd uses four windows: the *source window*, the *evaluation window*, the *program window*, and the *history window*.

The *source window* displays your program's source code. By moving the cursor within this window, you can tell **csd** which portion of code to evaluate, where to set tracepoints, and which stack frame to use.

The *evaluation window* is where **csd** accepts C expressions and variables that you type on the keyboard. When an expression is typed into this window, **csd** evaluates it and prints the result back in the evaluation window.

The *program window* is the area in memory where a program normally writes its output.

The *history window* is where **csd** records the events of your debugging session. Each time **csd** stops your program at a tracepoint, the traced statement or expression is written onto the history window. This is **csd**'s way of logging statements that it encounters during a debugging session.

Program window

To see the program window, type **<F7>**. **csd** normally jumps to the program window while your program is running, so you can see the output that your program has generated. This screen appears just as it would if your program were executed without **csd**.

You can choose to display a screen or window other than the program window while your program is running. To do so, use the **<F2>** key. Keep in mind that the program's output will go to the selected screen when you run the program. To erase the output, press **<F7>**, followed by the key for the selected screen or window.

Source window

The source window contains the source code of the program you are debugging. The cursor appears in this window when you first invoke **csd**. To return to this window from another screen or window, type the **<F8>** key.

If your program is more than one screen long (about 19 lines), not all of your program can be displayed at one time. To scroll through your program one line at a time, use the <↑> and <↓> arrow keys. The <PgDn> and <PgUp> keys move through your source program one **page** (or screen full of text) at a time. Finally, the <Home> and <End> keys jump to the beginning or the end, respectively, of your entire source file.

If you want to look at a line that contains a specific string, use the **<F1>** key. This provides a powerful search feature with which you can find a specific string and locate the next statement to be traced. The **<F1>** key is discussed in more detail later in this section.

To execute your program, press the **<F4>** key, then one of the following:

<F3>	Execute to the next tracepoint.
<return>	Execute one line of code.
<↓>	Same as <return>, but function calls are treated as one statement.
<←>	Execute to the end of the current function.
<Home>	Reload and initialize program.
<End>	Execute to the end of the program; do not stop at tracepoints.

If you have been moving through your source window and have lost track of what the next executable statement is, press the **<Shift-F8>** key. This will automatically position the cursor at the next executable statement.

When **csd** stops at a tracepoint in your source program, the cursor is positioned at the beginning of the traced source line, which is the next line to be executed.

csd displays both the source window and the evaluation window at the same time. The boundary between them is a reverse-video line that names the source file being displayed. If your program uses more than one source file, the name will change as you scroll from one file to another. **csd** displays the source files in the order in which you entered them on the **cc** command line.

You can change the relative sizes of the evaluation and source windows with the **<F2>** key, which will be described later in this section.

Evaluation window

The evaluation window is beneath the source window, under the reverse-video line. To jump to the evaluation window, press the **<F9>** key. This window is where you can type expressions that you want **csd** to evaluate. For example, if the variable **index** is an **int** that is in the current scope of the program and has the value '1', you can display its value by typing

```
index
```

into the evaluation window. **csd** will evaluate **index** and print the following:

```
index :: 1
```

Values are displayed in their natural format except for **structs** and unions, which are displayed as a list of hexadecimal bytes. For large structures, **csd** will display as much as will fit on the current line. For purposes of tracing, large structures are, of course, evaluated in their entirety. Strings are printed within quotation marks, and characters are enclosed within apostrophes.

You can type any legal C expression into the evaluation window. **csd** will check the legality of the expression as you type it. If you make an error in syntax or if a variable name is typed incorrectly, **csd** will give you an error message. To delete a character that is mistyped, use the backspace key. Typing **<ctrl-U>** deletes an entire line of text.

csd can evaluate expressions that use the following elements:

operators
constants (except '#' defined constants)
variables (in the current scope)
functions

In addition, the following special casts allow you to display information in a useful form:

oct octal
hex hexadecimal
str character strings

For example, if you issue the command

```
csd factor 12 22 32
```

then execute one statement by typing **<F4>** and **<return>**. Now you can enter the evaluation window, and type the following statements:

```
argv[1]
(str)argv[1]
(oct)argv[1]
```

csd will fill the evaluation window as follows:

```
argv[1] :: 0x5321
(str)argv[1] :: "12"
(oct)argv[1] :: 051441
```

Note that pointers are automatically displayed in hexadecimal. Keep in mind that **argv[1]** contains a pointer allocated by the run-time startup library routine and might have a different value for your system.

csd cannot evaluate expressions that contain C keywords, braces, labels, or semicolons. One exception to this rule is the keyword **sizeof**: it can be used in the evaluation window. Also, preprocessor information is not available in the evaluation window: **csd** does not recognize '#' defined symbols.

Expressions are evaluated as soon as they are typed in, and also whenever your program encounters a tracepoint, provided they are in the current scope.

Another powerful feature of **csd** is its ability to set tracepoints on expressions entered in the evaluation window. This causes **csd** to halt execution of your program if the value of the traced expression changes. To do this, **csd** enters single-step mode to evaluate all of the expressions in the evaluation window before each line of source code is executed.

To set a tracepoint on an expression in the evaluation window, switch to the evaluation window; then type the expression to be evaluated, but instead of typing **<return>**, type **<F3>**. The expression will shift into high-intensity characters to show that it is being traced. From now on, when you run the program in trace mode, execution will stop whenever the value of the traced expression changes.

When you type in expressions that contain variables, the variables must be in the current scope. This means that the variables that you are entering must be either external or declared by the function within which the cursor is positioned. Expressions that refer to several different scopes can appear in the evaluation window at the same time; however, **csd** cannot evaluate all of them at the same time. **csd** stops execution when it encounters a tracepoint and checks the expressions within the evaluation window; it evaluates only the expressions that are within the current scope.

Note that evaluation window expressions that produce side effects, such as assignments or **printf** statements, should be used with caution and deleted after use. Remember, too, that if you are tracing an expression, each expression in the window is evaluated after *each* statement in the program. Because the debugger needs to halt the program after each line to evaluate any traced expressions, execution is slowed considerably.

You can manipulate the evaluation window by using the following keys:

csd C source debugger

Up <↑>	Move up to previous expression
Down <↓>	Move down to next expression
Page Up <PgUp>	Move to previous page of expressions
Page Down <PgDn>	Move to next page of expressions
Begin <Home>	Move to beginning of evaluation window
End <End>	Move to end of evaluation window
Find <F1>	Move to pattern specified
Delete <Del>	Delete portions of the evaluation window
Insert <Ins>	Add expression at cursor

Please note the following:

1. Preprocessor information is not available in **csd**.
2. Literal strings may be used only as function arguments; you may only reference objects in your own data space. You can affect one assignment, as follows:

```
cp=strcpy(malloc(4), "cat")
```

3. Structure initializors, for example **{0, NULL, 135}**, are not expressions. Structures must be initialized one member at a time.

History screen

The history window is called by striking the <F10> key. This screen displays a log of events that occur while debugging, including every line of code or expression that was executed or traced while it had a tracepoint set.

You can manipulate the history window using the same keys as above, except **Insert**.

Up <↑>	Move up to previous line
Down <↓>	Move down to next line
Page Up <PgUp>	Move to previous page of lines
Page Down <PgDn>	Move to the next page of lines
Begin <Home>	Move to beginning of history window
End <End>	Move to end of history window
Find <F1>	Move to pattern specified
Delete <Del>	Delete portions of history window

Because the history window and the other I/O activity done by **csd** share the same buffers in memory, the contents of the history window are deleted automatically by **csd** as memory gets full.

You can redirect history window entries to a printer by using the **<F2>** command with the '**L**' option. Keep in mind that a program with many traced statements and expressions will generate large amounts of output.

Command keys

csd supports a number of different keystrokes for any given command. The following table lists each **csd** function and cursor movement command and the corresponding keys to invoke the command. The following sections describe the cursor movement commands.

FUNCTION	DESCRIPTION	COMMAND
Find	Find a string of text	<F1> or <esc>1 or <ctrl-S> or <esc>S or <ctrl-R> or <esc>R
Select	Select an option	<F2> or <esc>2
Trace	Trace an expression	<F3> or <esc>3
Run	Run the program	<F4> or <esc>4
Cancel	Cancel the last command	<F5> or <esc>5 or <F5>
Help	Display a help screen	<F6> or <esc>6 or <F6>
Program	Display program window	<F7> or <esc>7
Source	Display source window	<F8> or <esc>8
Evaluation	Enter evaluation window	<F9> or <esc>9
History	Display history window	<F10> or <esc>0
Up	Move cursor up	<↑> or <ctrl-P>
Down	Move cursor down	<↓> or <ctrl-N>
Out	Move to calling function	<←> or <ctrl-B>
In	Undo an **Out**	<→> or <ctrl-F>
Exit	Exit **csd**	<Shift-F1> or <ctrl-X> <ctrl-C>
Current	Return to current line	<Shift-F8> or <ctrl-X>X
Page Up	Move cursor up a page	<PgUp> or <esc>V
Page Down	Move cursor down a page	<PgDn> or <ctrl-V>
Delete	Delete a line	<Del> or <ctrl-K>
Insert	Insert a line	<Ins> or <ctrl-O>
Beginning	Beginning of source	<Home> or <esc><
End	End of source	<End> or <esc>>

Begin <Home>

The <Home> key can be used by itself to move the cursor or with another key to modify that key's action.

<Home>
Move the cursor to the beginning of the source, evaluation, or history windows. This command positions the cursor on the first line of the material in the screen or window.

<Del> <Home>
In the evaluation window or history window, remove all material from the beginning to the current cursor position.

<F1> <Home>
In the source window, the evaluation window, or the history window, search for a pattern from the current cursor location back toward the *beginning* of the window. In the case of the source window, *all* source files are searched.

<F4> <Home>
Reload and reinitialize the target program.

End <End>

The <End> key can be used by itself to move the cursor, or with another key to modify that key's action.

csd C source debugger

<End> Move the cursor to the last line of the source window, the evaluation window, or the history window.

<Del> <End>
Erase all lines from the current cursor position to the end of the window or screen. This works in the evaluation and history windows only.

<F1> <End>
In the source window, the evaluation window, or the history window, search for a pattern from the current cursor location through the end of the window. In the case of the source window, *all* source files are searched.

<F4> <End>
Execute the target program to the end, without stopping at tracepoints. All tracepoints, however, are logged into the history window.

Up <↑>

The <↑> key can be used by itself to move the cursor, or with another key to modify that key's action.

<↑> Move the cursor to the previous line in the source window, the evaluation window, or the history window.

<Del> <↑>
Erase the current line and move the cursor to the previous line. This works in the evaluation and history windows only.

<F1> <↑>
In the source, evaluation, or history windows, search for a pattern from the current cursor location through the beginning of the window. In the case of the source window, only the current source file is searched, beginning at the current cursor position and moving toward the first line of the current source module.

<F2> <↑>
In the source or evaluation window, increase the size of the evaluation window by one line and decrease the size of the source window by one line.

Down <↓>

The <↓> key can be used either by itself to move the cursor, or with another key to modify that key's action.

<↓> Move to the next line in the source window, the evaluation window, or the history window.

<↓> <Del>
In the evaluation window or the history window, remove the current line and move the cursor to the next line.

<F1> <↓>
In the source window, the evaluation window, or the history window, search for a pattern from the current cursor location through the end of the window. In the source window, search only the current source file.

<F2> <↓>
In the source or evaluation windows, decrease the size of the evaluation window by one line and increase the size of the source window by one line.

<F4> <↓>
Execute one line of code. A function call is treated as one single line of code. This is in contrast to the sequence **<F4> <return>**, which executes a single statement, even if the next

statement is within a called function.

Out <←>

The <←> key is used in the source window to change the current scope of a variable. When your program stops at a tracepoint, the active scope is that of the function being executed. Using the <←> key activates the scope of the function that called the current function. Then you can examine or change variables defined in the scope of the calling routine as well. When the scope is changed, the cursor is moved to the line of code that called the current function. Repeatedly pressing the <←> key will bring you to the outermost stack frame.

In <→>

When used in the source window, the <→> key undoes the effect of the <←> key. If you have not typed the <←> key since the last tracepoint, the <→> key has no effect.

Page Up <PgUp>

When used in the source window, the evaluation window, or the history window, the <PgUp> key scrolls the current window one page (or screen full of text) toward the beginning. If the current page of the source window is less than one page from the beginning of the file, the cursor will be moved to the beginning of the file.

If the cursor is positioned in the source window near the beginning of a source file, the <PgUp> key will move the cursor into the previous file, should there be one.

Page Down <PgDn>

In the source window, the evaluation window, or the history window, pressing the <PgDn> key moves the cursor one page toward the end. If the cursor is less than one page from the end, it will be moved to the end of the file.

If the cursor is in the source window and positioned near the end of a source file, the <PgDn> key will move the cursor into the next file, if there is one.

Find <F1>

The **<F1>** key searches for a specific pattern. For this reason, it is also called the *find* key.

The action of the **<F1>** key can be modified by a number of different keys, as follows.

<F1> <F3>
In the source and evaluation windows, find the next statement that has a tracepoint set on it.

<F1> <Home>
In the evaluation window and the history window, search toward the beginning. In the source window, search toward the beginning of the window and examine all files.

<F1> <↑>
In the evaluation window and the history window, search toward the beginning. In the source window, search toward the beginning but search only the current source file.

<F1> <End>
In the evaluation window and the history window, search toward the end. In the source window, search toward the end and examine all files.

<F1> <↓>
In the evaluation window and the history window, search toward the end. In the source window, search toward the end, but search only the current source file.

csd C source debugger

The string that **<F1>** seeks is a *pattern* of characters. These characters can be ordinary alphanumeric characters, or a mixture of alphanumeric characters with *wildcard* characters that modify how the search is conducted.

The simplest pattern is a set of non-special characters, which is matched literally.

The following special characters can be used to specify powerful patterns:

^	Match only the beginning of a line
$	Match only the end of a line
?	Match any one character
*	Match any number of characters
\	Escape character: use it to search for a wildcard literally
[*abc*]	Match any one of *a*, *b* or *c*
[*a-m*]	Match any of the letters *a* through *m*

To match a line in the source, evaluation, or history windows that ends in a ';', type the sequence

<F1>;$<return>

This says to find a pattern that has a semicolon followed by the end of a line.

Keep in mind that **<F1>** will search for the strings exactly as they appear on the screen. For example, to find the line

```
a :: 11
```

where **11** is the value supplied by **csd**, type:

<F1>1$<return>

If you want to search for any of the special characters of the patterns, precede that character by a backslash '\'. For example, to find the line

```
c = a*b;
```

type:

```
a\*b
```

Note that if you do not type a pattern after the **<F1>** key, **csd** will search for the last pattern used, if there is one.

If you start an **<F1>** command and wish to abandon it or restart it, press the **<F5>** key. Typing **<ctrl-U>** restarts the find pattern.

Insert <Ins>

The <Ins> command is used only in the evaluation window. It opens a blank line so you can enter a new expression above the one entered in the current line.

Delete <Del>

The <Del> key can be used in the evaluation window or the history window to remove or "kill" text. Its action can be modified by other keys, as follows:

<Del> <↑>
Kill the current line of text, and reposition the cursor on the previous line.

<Del> <↓>
Kill the current line of text, and reposition the cursor on the following line.

<Del> <Home>
Kill all text from the beginning of the file to the cursor.

<Del> <End>
Kill all text from the cursor to the end of the file.

To cancel a kill command before it is executed, type **<F5>**.

You should periodically use **<Del>** to purge the history window of information you no longer need. If you do not, it will take up space that could be used to hold source lines in memory, and slow down **csd**.

Run <F4>

The **<F4>** key, which can be used in any window, tells **csd** to execute your program. Its operation is modified by the following keys.

<F4> <F3>
Execute the program up to the next tracepoint. The tracepoint can be either in the source window, in which case execution stops just before execution of the traced line, or in the evaluation window, in which case **csd** stops if the value of the traced expression changes. The cursor is moved to the statement or expression that bears the tracepoint, unless it is in a different window.

<F4> <return>
Execute one line of code. If that line is a function call, execution stops on the first line of that function.

<F4> <↓>
Execute one line of code, as with the **<return>** modifier discussed above, but treat function calls as a single line of code.

<F4> <→>
Execute to the end of the current function.

<F4> <Home>
Reload and restart the program being debugged.

<F4> <End>
Execute the program to its end, without stopping at any traced statements or expressions. Traced statements and expressions will be logged onto the history screen.

To cancel **<F4>,** press the **<F5>** key.

Trace <F3>

The **<F3>** sets a tracepoint on the line on which the cursor is currently positioned. Traced lines are written in high-intensity characters. To turn a tracepoint off, simply move the cursor back to the line being traced and press **<F3>** again. Tracepoints can be set either in the source window or in the evaluation window.

When a tracepoint is set on a line of code in the source window, execution stops immediately *before* that line of code. Only executable lines of code can be traced; statements such as comments and declarations cannot be traced. Note, too, that because the compiler optimizes unused and some common code out of existence, some things that appear to be executable actually cannot be executed. If you try to set a tracepoint on a line that has no executable code, the debugger will give you the error message:

```
not executable statement
```

When a tracepoint is set on an expression in the evaluation window, execution of the program stops when the value of the traced expression changes.

Tracing an evaluated expression causes **csd** to single-step through the entire program. This is done so **csd** can recognize the change in the variable as soon as possible. Thus, tracing an expression in the evaluation window noticeably slows the execution of the program being debugged. The execution will also be noticeably slower if a large number of statements are executed if you Run to the next line, or Run to the end of the function.

Select <F2>

The **<F2>**, or "select", key controls the way **csd** presents its screens. How it functions is controlled by a number of modifying keys, as follows.

<F2> <↑>
Move the boundary between the source and evaluation windows up by one line.

<F2> <↓>
Move the boundary between the source and evaluation windows down by one line.

<F2> <F7>
Display the program's output on the program window. This is the default.

<F2> <F8>
Display the program's output in the source window. The source window will shift to display the code currently being executed. Note that if no tracepoints are set, execution may be too fast to follow.

<F2> <F9>
Display the program's output in the evaluation window. The variables in the evaluation window will change as the program executes. Note that if no tracepoints are set, execution may be too fast to follow.

<F2> <F10>
Display the program's output on the history window.

<F2> L Redirect to the printer all material normally logged onto the history window. Pressing this combination of keys again turns off this feature.

To cancel an **<F2>** command, press the **<F5>** key.

Help screens

If you have a question about how the debugger operates, press **<F6>**. This will let you invoke any of the following help screens to refresh your memory about how a given **csd** command works.

Note that the windows may appear slightly differently on your screen due to differences in formatting.

General help

Program <F7>
Enter program window. This is the window that is usually shown while the program is running.

Source <F8>
Enter source window, where the program source is shown.

Eval <F9>
Enter evaluation window, where C expressions are evaluated.

History <F10>
Enter the history window, where a log of traced statements and expressions is shown.

Help <F6>
Display helpful information, or return to debugger if already in the help window.

Exit <Shift-F1>
Exit the debugger.

The keys below are not accepted in the program window.

Beg<Home>
Go to beginning of display; in the source window, this is the first line of the first source file.

End <End>
Go to the end of the display; in the source window, this is the last line of the last source file.

Out <←>
In the source window, go to the statement from which the current function was called.

In <→>
Go back to the statement from which you went **Out**.

Move about with **<↑> <↓>** (up, down arrows).

For more on the following commands, press the corresponding key.

Trace <F3> **Run <F4>** **Eval <F9>** **Find <F1>**
Insert <Ins> **Delete <Del>** **Select <F2>**

Trace

In the **source window**, the current statement is marked traced. The statement is shown highlighted, and whenever it is executed it is first copied to the history log. If the statement is already traced, the trace is removed.

Only executable statements may be traced. This excludes all comments and declarations. Furthermore, the optimizer in the compiler deletes unreachable and duplicated code (usually break, continue or return statements); deleted code is not executable.

In the **evaluation window**, the current expression is traced. This means that before each source statement is executed, the expression is re-evaluated and is copied to the history log whenever it is found to have changed value. If the expression is already traced, the trace is removed.

This command is not accepted in any other window.

The program may also be run until a trace is encountered: see **Run <F4>**.

Press **Help <F6>** to return to the debugger.

Execution

Run <F4>

Execute program.

The next key must be one of the following:

Trace <F3>

Program executes until a traced statement is encountered or a traced expression changes value.

Return <return>

One statement is executed (i.e., program is single-stepped).

Down <↓>

One statement in the current function is executed. A function call is treated as an indivisible statement.

Beg <Home>

Program is reloaded and execution is reinitialized.

End <End>

Program executes through to the end, logging tracepoints in history window.

Out <←>

Program executes until the current function returns.

Cancel <F5>

Cancels the **<F4>** command.

The program executes at full speed if there are no traced expressions in the evaluation window (traced statements do not significantly impact the speed) and if you use **Run Trace** or **Run End**. Otherwise, it runs at single-step speed, which can be significantly slower.

Press **Help <F6>** to return to the debugger.

Evaluation window

Evaluating C Expressions

In the evaluation window, a C expression may be entered involving operators, literal strings, constants and variables in the current scope (the scope of the current line in the source window). An expression may **not** involve keywords (such as **if**, **for**) or labels or braces '{}'or semi-colons ';'. See your C reference manual for details.

Expressions are evaluated immediately, and whenever the program encounters a tracepoint. The resulting value is displayed in a format appropriate to the type. No value is displayed if the expression involves an automatic variable and the current function does not have an active stack frame. The debugger defines types **(oct)**, **(hex)** and **(str)** for use as casts to display values in base 8, base 16, or as character strings, respectively. Macro names (generated by **#define**) are not recognized.

Expressions with side effects (assignments, increments, decrements, functions performing I/O) will naturally cause these effects whenever evaluated; they should probably be removed after being evaluated the first time. Literal character strings may be used as arguments to functions (including **strcmp** and **strcpy**), but cannot be assigned to variables, since the space for any literal in a C program is allocated statically.

Press **Help <F6>** to return to the debugger.

Find

Find <F1>

Find the first line in the current window matching a pattern, which may be the Trace key or metacharacters.

Trace <F3>

The **Trace** key causes the pattern to be highlighted and matches traced statements or expressions.

```
^       matches the beginning of a line.
$       matches the end of a line.
?       matches any character.
*       matches a string of zero or more of any characters.
[abc]   a class (in [ ]) matches any member of the class.
[i-n]   part of a class may be specified as a range.
```

If not given, the pattern defaults to the previous value.

The command is terminated by one of the following:

Up <↑>

Search up to the beginning of the current source file or top of the evaluation or history display.

Down <↓>

Search down to the end of the current source file or bottom of the evaluation or history display.

Beg <Home>

Search up to the beginning of the first source file or top of the evaluation or history display.

End <End>

Search down to the end of the last source file or bottom of the evaluation or history window.

Cancel <F5>

Cancel the **<F1>** command.

If the pattern is not found, this failure is reported.

Press **Help <F6>** to return to the debugger.

Insert/Delete

Insert<Ins>

Insert a blank line above the current line and make it current. This is only permitted in the evaluation window.

Delete <Del>

Delete a range of lines in the current window. This works in the evaluation and history windows only.

The command is terminated by one of the following:

Up <↑>

Delete the current line and move up one line.

Down <↓>

Delete the current line and move down one line.

Beg <Home>

Delete from the current line to the beginning of the display.

End <End>

Delete from the current line to the end of the display.

Cancel <F5>

Cancel the **<Del>** command.

If you get an "out of space" message in the evaluation window, it may be necessary to remove part of the display to free space for new expressions. If the debugger is rereading the disk for each new screen of source, removing several lines of history or evaluation may free enough space for more source buffers, eliminating the extra disk activity.

Press **Help <F6>** to return to the debugger.

Select

Select <F2>

Select various auxiliary options.

The command is terminated by one of the following:

Up <↑>

Move the boundary between the source and evaluation windows up one line.

Down <↓>

Move the boundary between the source and evaluation windows down one line.

L

Toggle the listing switch (initially off). When on, this copies the history log to the printer.

Program <F7>

Select the program window to be displayed when the program is running. When initially invoked, the debugger selects the program window by default.

Source <F8>

Select the program window to be displayed when the program is running. Any program output will be displayed in the source window.

Eval <F9>

Select the evaluation window to be displayed when the program is running. Any program output will be displayed in the evaluation window.

History <F10>

Select the history window to be displayed when the program is running. Any program output will be displayed in the history window.

Cancel <F5>

Cancel the **<F2>** command.

Press **Help <F6>** to return to the debugger.

History

History log

The history window stores messages about the program being debugged.

These messages begin with csd's sign on banner, the list of source files found in the debug tables, and an announcement that csd has loaded the program for execution.

If the program causes a machine exception, the details are reported. If the program terminates normally, the exit status is reported. If you **Run** the program after a normal or abnormal termination, the reload for execution is reported. If you **Run** the program to the **End**, **Out**, or **Down**, traced statements and changes in the values of traced expressions are logged to the history window as they occur. If csd encounters an error during the session, the details are reported.

You can use the cursor movement and **Find** commands to examine the history log, and you can **Delete** lines.

csd automatically discards history, from the beginning, when other commands for memory begin to fail.

Press **Help** to return to the debugger.

Cursor movement

FUNCTION	KEY	ACTION
Begin	<Home>	Go to beginning of display; in the source window this is the first line of the first source file.
End	<End>	Go to end of the display; in the source window this is the last line of the last source file.
Fbegin	<ctrl-A>	Go to beginning of current source file.
Fend	<ctrl-E>	Go to end of the current source file.
Up	<↑>	Move the cursor one line up.
Down	<↓>	Move the cursor one line down.
Previous	<ctrl-↑>	Move the cursor one page up.
Next	<ctrl-↓>	Move the cursor one page down.
Out	<←>	In the source window go to the statement from which current function was called.
In	<→>	Go back to the statement from which you went **Out**.
Locate	<Home>	Go to the source statement at which execution is stopped.

Cursor movement is possible in the source, evaluation, and history windows.

Press **Help** to return to the debugger.

Error messages

The following are the error messages printed by **csd**. All **Message**s are displayed in the history window. All **Fatal error** messages are displayed in the history window. Fatal errors force the history window to be displayed, and ask that you exit from **csd**.

usage: CSD -G -D -O[L][M] -Hhelppath -Ssourcepath -T file[.exe] args
: Fatal error. You have specified improper arguments to **csd**. This message tells you the form of the **csd** command. If the source file is to be found other than on the current disk or directory, you need to specify the **-S** option to tell **csd** where to find it. If the **csd** disk is not in the current disk drive or directory, you need to tell **csd** where it is with the **-H** option so that it can find the help files. The **-G** option is used when running **csd** on a color system which is not in 80 x 25 character mode. The **file** is the name of the program that you are debugging. Finally, **[args]** is the list of arguments to your program. For a complete listing of the compile command arguments and their uses, see section 6, **csd** *Command reference*.

Reading tables...
: Message. This means that the tables containing debugging information are being read in.

Loading *filename*...
: Message. *filename* is the name of the program being debugged. This message is printed the first time the program is loaded.

Reloading *filename*...
: Message. This message is printed when the program is reloaded after <Run><End>, <Run><Begin>, a machine fault or after the program exits under its own control.

Adding source: *sourcefile*
: After the "Reading tables..." message, you will see one of these for each source file found in the tables.

file **.exe** : cannot open
: Fatal error. The file that you have specified in the **csd** command cannot be found.

insufficient memory for csd to run
: Fatal error. Not enough memory is available for **csd** to load the symbol table and allocate the number of kilobytes specified with the **-D** option. Reduce the number of bytes specified.

insufficient memory to load program
: Fatal error. Not enough memory is left after loading the symbol table and the sources to run the program. Reduce the number of bytes specified with the **-D** option.

this source file is more recent than the program
: Message. This message reports that the program probably needs to be recompiled to bring it up to date with the source.

blocks nested too deeply
: Fatal error. The program has nested compound statement blocks deeper than **csd** can keep track of.

valid breakpoint traps are 0, 3,...12, and 15
: Fatal error. This reports that the -Itrap# option has an invalid value. The default value is 12.

csd stack overflow
: Message. This message appears when **csd** overflows its own stack.

junk in high byte of program counter: *addr*
: Message. This reports that the traced program has a program counter with non-zero value in the high 8 bits. These bits are not used by the 68000 processor, but it's unusual for them to be set.

no room in **vblqueue** for break key handler
: Message. This means that the break key will not be active. This should not appear in most circumstances because there are seven free slots available.

Can't read *file***.prg** read *file***.prg**'>=29
: Fatal error. **csd** has encountered a read error trying to read your program.

No source!
: Fatal error. No source files were compiled with the **-VCSD** option. You must use this option when compiling programs for debugging if you want to use them with **csd**.

out of memory
: Fatal error. There is not enough room to hold the program's debug tables. Try using the **-D** option to increase the size of **csd**'s workspace. Otherwise, debug smaller portions of your program.

execute failed: *filename*
: This means that the program cannot be executed.

*sourcefile***.c**: cannot open
: One of the source files used to compile your program cannot be found. The names of the source files are kept in the executable **.exe** file, so if you have moved them, **csd** cannot find them. Use the **-T** or **-S** command line option to tell **csd** where to find them.

Cannot Open
: This means that the file name specified as the program to be debugged cannot be opened to read the debug tables.

file has a corrupted image, should be recompiled
: Fatal error. In order to start tracing your program, **csd** must write a breakpoint instruction into the program file. When tracing begins, the program file is rewritten to its original value. This message means that something went wrong, and **csd** left the breakpoint in the program file. The program cannot be traced as it is, and you will have to relink from object files or recompile.

try Help
: You have pressed a key that is not acceptable at this time. For example, if you are at the beginning of the source file and type **<↑>**, you will get this message.

at outermost frame
: In exploring the stack, you have typed the **<←>** key, but your caller, if it exists, is not visible to **csd**.

at innermost frame
: In exploring the stack, you have pressed the **<→>** key, but you are already positioned at the innermost frame; that is, you have undone all of the **<←>**'s.

not executable statement
: You are trying to set a trace on a statement that is not executable. Declarations, comments, braces '{ }', and '#' statements are not executable. In addition, code for some statements is combined during optimization, so that statements such as **break**, **continue** and **do** may not have any executable code.

not traceable expression
: The expression you are trying to trace cannot be traced because it has no value. This is true of functions that are of **void** type.

out of space
: Remove some of the expressions, or delete some of the history window.

helpfile.**hlp**: cannot open
: **csd** cannot find *helpfile*.**hlp** in the current path(s). You can use the **-H** option to correct this.

more space required
: There is not enough room to add expressions to the evaluation window. Remove some expressions from the evaluation window and try again.

Can't move boundary any further move boundary any further'>=29
: You are trying to move the cursor so it is positioned before the beginning or after the end of the expression, history, source or evaluation windows. **csd** requires that the source window be at least three lines long, and that the evaluation window be at least two lines long.

exit status *nnn*
: Message. this message reports the value returned by the program when it terminated.

Evaluation window error messages
: The following messages refer to expressions in the evaluation window. They describe errors in entering expressions, and are self explanatory:

```
integer type required
integer or pointer type required
numeric type required
numeric or pointer type required
expression required
literal or variable in current scope required
struct or union member required
lvalue required
pointer required
lvalue or type required
terminal or prefix op required
function required
array or pointer required
pointer to struct or union required
struct or union required
infix operator required
matching '?' and ':' required
pointer to same type required
matching types required
')' required
']' required
```

Machine fault names

The following lists named machine faults.

bus error
: Access to an address which does not exist, or which the hardware forbids in user mode.

Address error
: Access to an odd address for a long or word operand.

Illegal instruction
: Execution of an instruction which is not supported by the processor.

Divide by zero
: The divide instruction was given a zero as a divisor.

Bounds trap
: The 68000 chk instruction trapped.

Overflow trap
: The 68000 trapv instruction trapped.

Index

 # to _

User Reaction Report

To keep this manual free of errors and to facilitate future improvements, we will appreciate receiving your reactions. Please fill in the appropriate sections below, detach, and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name:

Company:

Address:

Phone:

Date:

Version and hardware used:

Did you find any errors in the manual?

Can you suggest any improvements to the manual?

Did you find any bugs in the software?

Can you suggest improvements or enhancements to the software?

Additional comments:

OTHER QUALITY PRODUCTS FROM

THE MARK WILLIAMS COMPANY

LET'S C **$75.00**

Intrigued by the idea of programming in C? Let's C is a complete, high-quality C compiler which compiles fast and produces fast, compact code. PC TECH JOURNAL writes "The performance and documentation of the $75 Let's C compiler rival those of C compilers currently being sold for $500...." For IBM and compatibles.

LET'S C UTILITIES **$39.95**

This collection of UNIX programming tools will increase your productivity and streamline your programming. The Let's C Utilities include *ed*, the classic line editor; *diff*, which compares two files; *m4*, a powerful macro processor; *sort*, a versatile file-sorting utility; and *uniq*, which removes repeat lines in a file. Together with Let's C and CSD, these utilities provide a complete C programming system. For IBM and compatibles.

FAST FORWARD **$69.95**

Wish your computer could run faster? With Fast Forward it can! Fast Forward is a software utility that makes all your software run up to 10 times faster. By using your computer's high-speed internal memory (RAM), Fast Forward bypasses the slow disk drive. Programs requiring frequent disk access will show amazing improvements. For IBM and compatibles.

MARK WILLIAMS C FOR THE ATARI ST **$179.95**

If you do C programming on an Atari ST, this is the C compiler for you. It's faster than ever, and includes GEM documentation, a symbolic debugger, an integrated editor for easy bug fixes, and even a RAM disk. ANALOG COMPUTING called it "the all-around best choice for serious software development on the ST." Over a hundred sample programs make it ideal for novices, too.

60 Day Money Back Guarantee

For more information on any of these products, call (312) 472-6659

Order Form

You may order additional products from the Mark Williams Company with this convenient card. Simply fill out the form, then detach, fold and seal it. Postage is pre-paid, and we will ship your order the same day we receive your request.

Quantity	*Product*	*Unit Price*
_____	**Let's C** (IBM PC)	$75.00
_____	**csd C Source Debugger** (IBM PC)	$75.00
_____	**Let's C Utilities** (IBM PC)	$39.95
_____	**Fast Forward** (IBM PC)	$69.95
_____	**Mark Williams C** (Atari ST)	$179.95
_____	**COHERENT** (IBM PC)	$495.00
_____	**XYBASIC** (z80, 8080)	Call
_____	**C Cross-Compilers**	Call

If you would like information on any of these products, put a check next to the product above and then check the Information Only box below.

60 Day Money Back Guarantee

- - - - - - - - - - - - - - - - - fold along line - - - - - - - - - - - - - - - - -

Please clearly indicate the method of payment on all orders and provide your complete shipping address.

[ ] Check
[ ] Money Order
[ ] Information Only

[ ] Visa, Master Card, American Express
Card #: _____
Expiration Date: _____

Name _____

Address _____

City _____ State _____ Zip _____

Daytime Phone (_____) _____ - _____

Signature _____

Fold and seal order form before mailing.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS • PERMIT NO. 10820 • CHICAGO, IL

POSTAGE WILL BE PAID BY ADDRESSEE

Mark Williams Company
1430 Wrightwood Avenue
Chicago, Illinois 60614

MARK WILLIAMS COMPANY ("MWC")
Software License Agreement

YOU SHOULD CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT BEFORE BREAKING THE SEAL ON THE DISKETTE ENVELOPE. BREAKING THE SEAL INDICATES YOUR ACCEPTANCE OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE DISKETTE UNOPENED, AND YOUR MONEY WILL BE REFUNDED.

MWC provides this software and licenses its use to you. You assume responsibility for the selection of the software to achieve your intended results, and for the installation, use and results obtained from it.

LICENSE

MWC grants you a license only to: (a) use the software on a single machine; and (b) copy the software into any machine readable form for backup purposes in support of your use of the software on the single machine, unless the software includes mechanisms to limit or inhibit copying.

As an exception to the foregoing paragraph, we grant you the right to include portions of the MWC Runtime Library (as defined below) in software programs that you develop, called Composite Programs, and to use, distribute and license Composite Programs to third parties without payment of any further license fee. You shall, however, include in the object code for each Composite Program a notice in this form: "Portions of this program, copyright 1984-1987, Mark Williams Company." As an express condition to the use of the software, you agree to indemnify and hold MWC harmless from all claims by you and third parties arising out of the use of any Composite Program. Any portion of the Runtime Library merged into another program will continue to be subject to the terms and conditions of this Agreement. "Runtime Library" is defined as the set of copyrighted MWC language subroutines provided with the software, a portion of which must be linked to and become part of a Composite Program for that Program to run on a computer.

You may not use, copy or modify the software except as expressly provided for in this license. You may not transfer the software, or any copy, modification or merged portion, in whole or in part.

TERM

You may terminate the license at any time by destroying the software together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any terms or conditions of this Agreement. You agree upon termination to destroy the software together with all copies, modifications and merged portions in any form. The license shall terminate upon termination of this Agreement.

LIMITED WARRANTY

EXCEPT FOR THE LIMITED WARRANTY SET FORTH IN THE NEXT PARAGRAPH, THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT MWC OR ANY AUTHORIZED MWC DEALER) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. MWC DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE.

MWC warrants to the original licensee that the diskette(s) on which the software program is recorded is free from defects in material and workmanship under normal use and service for a period of 60 days from the delivery date as evidenced by a copy of your receipt. Your exclusive remedy is replacement of the defective diskette(s) provided that you return it to MWC with a copy of your receipt.

IN NO EVENT WILL MWC BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF MWC OR AN AUTHORIZED MWC DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

MISCELLANEOUS

You may not sublicense, assign or transfer the license to the software except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void. This Agreement will be governed by the laws of the State of Illinois.

Should you have any questions concerning this Agreement, you may contact MWC by writing to Mark Williams Company, 1430 W. Wrightwood Ave., Chicago, Illinois 60614.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

csd™

THE DEBUGGER THAT CUTS DEVELOPMENT TIME IN HALF!

A fast C compiler is great. For half the job. Then you're in the dark. Groping your way through clunky assembler or long hex dumps. Poking around for those program-clogging bugs.

You need *csd*! The revolutionary source level debugger from Mark Williams. With *csd*, you'll actually debug in C. It's like suddenly having your eyes opened. Four separate windows give you a clear view. See your original code. Your program's output. The value of any variable or expression. And a log of traced expressions and statements.

The result? Experienced programmers get programs finished faster. Beginners learn by watching and doing instead of by guessing. And everybody cuts overall development time in half.

csd is designed exclusively for use with the Let's C compiler. Both are from Mark Williams Company, makers of quality programming tools since 1976.

FEATURES

- For use with Let's C compiler
- Debugs in C source code, not assembler
- Cuts development time in half
- Helpful learning tool
- Large and small memory model
- Provides 4 separate windows:
  - * Source–your original code
  - * Program–your program's output
  - * Evaluation–value of variables and expressions
  - * History–log of traced expressions and statements
- Offers on-line help screens
- No knowledge of machine architecture or assembler necessary
- Does not change program's speed or size
- Monitors variables while tracing programs
- Can interactively evaluate any C expression
- Can execute any C function in your program
- Ability to set trace points on variables or expressions
- Not copy protected

What the reviewers say:

"csd *is close to the ideal debugging environment…a definite aid for learning C and an indispensable tool for program development.*"
– William G. Wong, BYTE

"This is a powerful and sophisticated debugger built on a well-designed, 'serious' compiler."
– Jonathon Sachs, Micro/Systems Journal

"…a boon to people who are learning C."
– Kaare Christian, PC MAGAZINE

csd runs with Let's C on IBM PCs or 100% compatibles with MS-DOS 2.0 or greater. *csd* requires 384K RAM and two double-sided IBM compatible 320 disk drives or hard disk. Let's C is a registered trademark and *csd* is a trademark of the Mark Williams Company.

ISBN 0-927860-04-X