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.

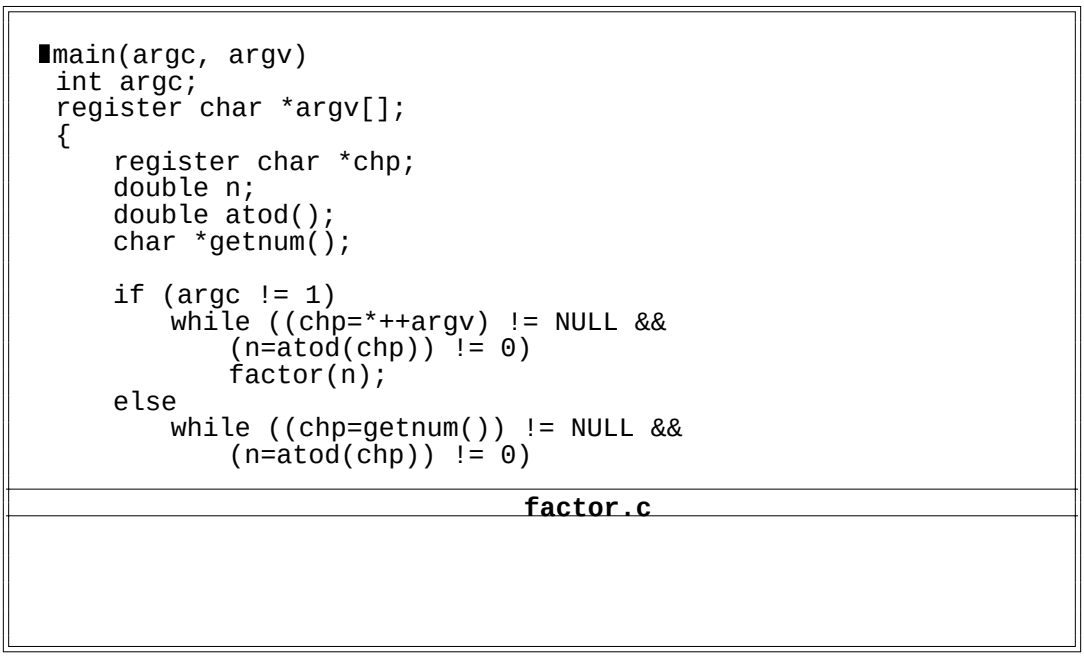
csd evaluates the expressions you enter in the evaluation window when you first type them in and press **<return>**. It also evaluates the expressions you have typed in the window whenever the program stops executing--that is, when it reaches the end of the program, reaches a tracepoint, or when you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



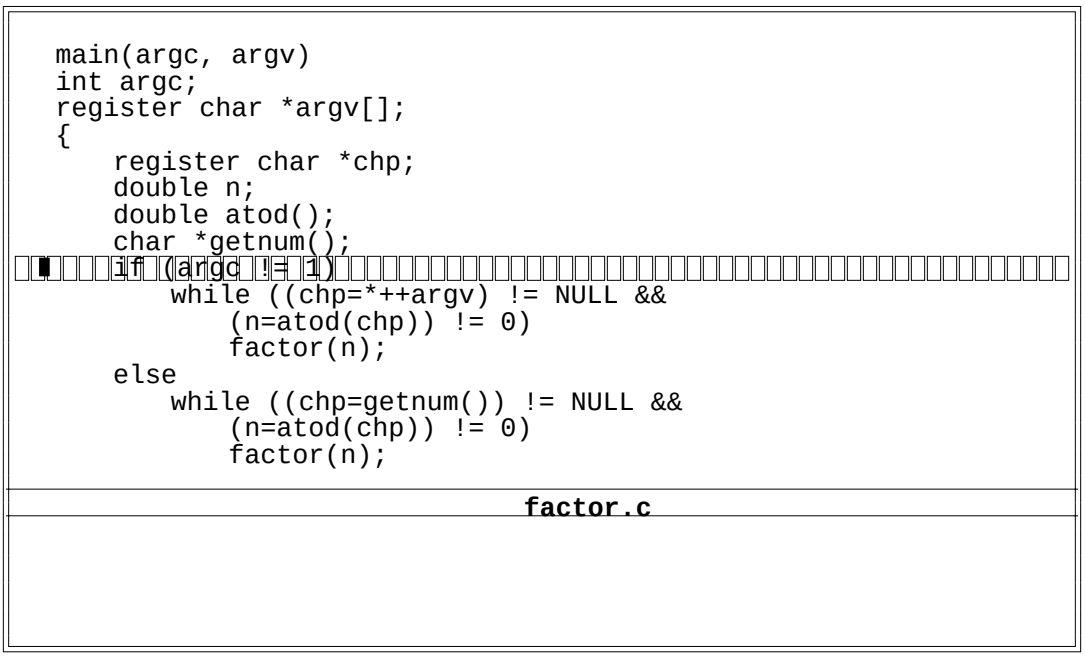
Press the cursor-movement key **<↓>** or **<ctrl-N>**, until the cursor is at the line

```
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
```

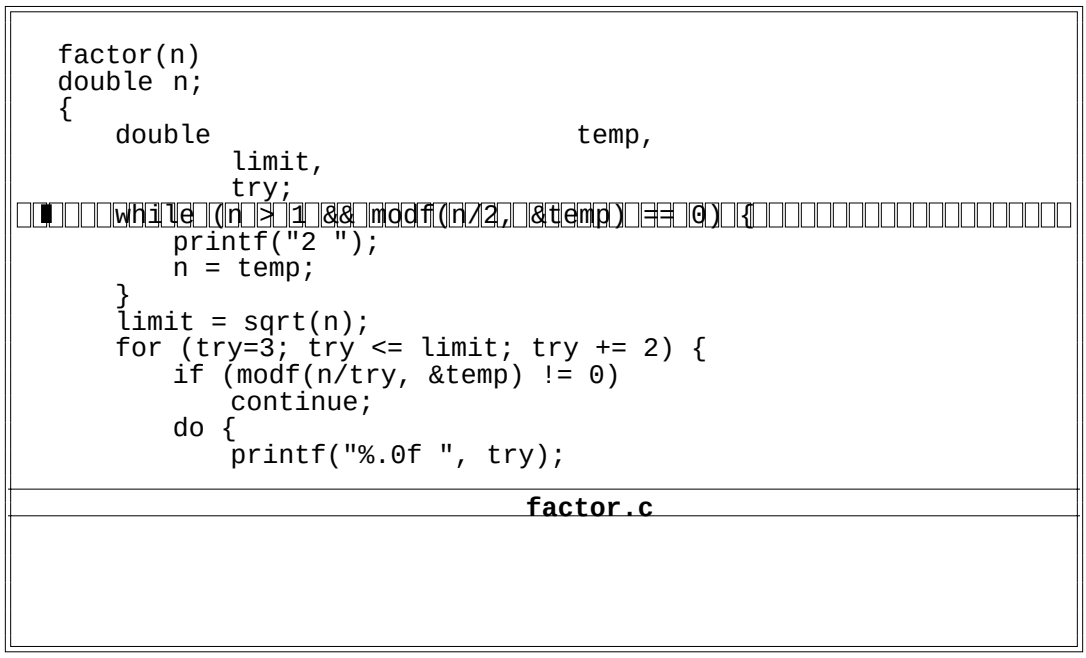
You can easily find the *function* **factor** within the program. Type **<F1>** followed by

```
^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.