

**D B U G**  
**C Program Debugging Package**

**by**

*Fred Fish*

## INTRODUCTION

Almost every program development environment worthy of the name provides some sort of debugging facility. Usually this takes the form of a program which is capable of controlling execution of other programs and examining the internal state of other executing programs. These types of programs will be referred to as external debuggers since the debugger is not part of the executing program. Examples of this type of debugger include the **adb** and **sdb** debuggers provided with the **UNIX**<sup>1</sup> operating system.

One of the problems associated with developing programs in an environment with good external debuggers is that developed programs tend to have little or no internal instrumentation. This is usually not a problem for the developer since he is, or at least should be, intimately familiar with the internal organization, data structures, and control flow of the program being debugged. It is a serious problem for maintenance programmers, who are unlikely to have such familiarity with the program being maintained, modified, or ported to another environment. It is also a problem, even for the developer, when the program is moved to an environment with a primitive or unfamiliar debugger, or even no debugger.

On the other hand, *dbug* is an example of an internal debugger. Because it requires internal instrumentation of a program, and its usage does not depend on any special capabilities of the execution environment, it is always available and will execute in any environment that the program itself will execute in. In addition, since it is a complete package with a specific user interface, all programs which use it will be provided with similar debugging capabilities. This is in sharp contrast to other forms of internal instrumentation where each developer has their own, usually less capable, form of internal debugger. In summary, because *dbug* is an internal debugger it provides consistency across operating environments, and because it is available to all developers it provides consistency across all programs in the same environment.

The *dbug* package imposes only a slight speed penalty on executing programs, typically much less than 10 percent, and a modest size penalty, typically 10 to 20 percent. By defining a specific C preprocessor symbol both of these can be reduced to zero with no changes required to the source code.

The following list is a quick summary of the capabilities of the *dbug* package. Each capability can be individually enabled or disabled at the time a program is invoked by specifying the appropriate command line arguments.

- o Execution trace showing function level control flow in a semi-graphically manner using indentation to indicate nesting depth.
- o Output the values of all, or any subset of, key internal variables.
- o Limit actions to a specific set of named functions.
- o Limit function trace to a specified nesting depth.
- o Label each output line with source file name and line number.
- o Label each output line with name of current process.
- o Push or pop internal debugging state to allow execution with built in debugging defaults.
- o Redirect the debug output stream to standard output (stdout) or a named file. The default output stream is standard error (stderr). The redirection mechanism is completely independent of normal command line redirection to avoid output conflicts.

---

1. UNIX is a trademark of AT&T Bell Laboratories.

## PRIMITIVE DEBUGGING TECHNIQUES

Internal instrumentation is already a familiar concept to most programmers, since it is usually the first debugging technique learned. Typically, "print statements" are inserted in the source code at interesting points, the code is recompiled and executed, and the resulting output is examined in an attempt to determine where the problem is.

The procedure is iterative, with each iteration yielding more and more output, and hopefully the source of the problem is discovered before the output becomes too large to deal with or previously inserted statements need to be removed. Figure 1 is an example of this type of primitive debugging technique.

```
main (argc, argv)
int argc;
char *argv[];
{
    printf ("argv[0] = %d\n", argv[0]);
    /*
     *      Rest of program
     */
    printf ("== done ==\n");
}
```

Figure 1  
Primitive Debugging Technique

Eventually, and usually after at least several iterations, the problem will be found and corrected. At this point, the newly inserted print statements must be dealt with. One obvious solution is to simply delete them all. Beginners usually do this a few times until they have to repeat the entire process every time a new bug pops up. The second most obvious solution is to somehow disable the output, either through the source code comment facility, creation of a debug variable to be switched on or off, or by using the C preprocessor. Figure 2 is an example of all three techniques.

```

int debug = 0;

main (argc, argv)
int argc;
char *argv[];
{
    /* printf ("argv = %x\n", argv) */
    if (debug) printf ("argv[0] = %d\n", argv[0]);
    /*
     *      Rest of program
     */
#ifdef DEBUG
    printf ("== done ==\n");
#endif
}

```

Figure 2  
Debug Disable Techniques

Each technique has its advantages and disadvantages with respect to dynamic vs static activation, source code overhead, recompilation requirements, ease of use, program readability, etc. Overuse of the preprocessor solution quickly leads to problems with source code readability and maintainability when multiple **#ifdef** symbols are to be defined or undefined based on specific types of debug desired. The source code can be made slightly more readable by suitable indentation of the **#ifdef** arguments to match the indentation of the code, but not all C preprocessors allow this. The only requirement for the standard **UNIX C** preprocessor is for the '#' character to appear in the first column, but even this seems like an arbitrary and unreasonable restriction. Figure 3 is an example of this usage.

```

main (argc, argv)
int argc;
char *argv[];
{
#   ifdef DEBUG
    printf ("argv[0] = %d\n", argv[0]);
#   endif
    /*
     *      Rest of program
     */
#   ifdef DEBUG
    printf ("== done ==\n");
#   endif
}

```

Figure 3  
More Readable Preprocessor Usage

## FUNCTION TRACE EXAMPLE

We will start off learning about the capabilities of the *debug* package by using a simple minded program which computes the factorial of a number. In order to better demonstrate the function trace mechanism, this program is implemented recursively. Figure 4 is the main function for this factorial program.

```
#include <debug.h>

int main (argc, argv)
int argc;
char *argv[];
{
    int result, ix;
    extern int factorial(int);
    DEBUG_ENTER ("main");
    DEBUG_PROCESS (argv[0]);
    for (ix = 1; ix < argc && argv[ix][0] == '-'; ix++) {
        switch (argv[ix][1]) {
            case '#':
                DEBUG_PUSH (&(argv[ix][2]));
                break;
        }
    }
    for (; ix < argc; ix++) {
        DEBUG_PRINT ("args", ("argv[%d] = %s", ix, argv[ix]));
        result = factorial (atoi(argv[ix]));
        printf ("%d\n", result);
    }
    DEBUG_RETURN (0);
}
```

Figure 4  
Factorial Program Mainline

The **main** function is responsible for processing any command line option arguments and then computing and printing the factorial of each non-option argument.

First of all, notice that all of the debugger functions are implemented via preprocessor macros. This does not detract from the readability of the code and makes disabling all debug compilation trivial (a single preprocessor symbol, **DEBUG\_OFF**, forces the macro expansions to be null).

Also notice the inclusion of the header file **debug.h** from the local header file directory. (The version included here is the test version in the *debug* source distribution directory). This file contains all the definitions for the debugger macros, which all have the form **DEBUG\_XX...XX**.

The **DEBUG\_ENTER** macro informs that debugger that we have entered the function named **main**. It must be the very first "executable" line in a function, after all declarations and before any other executable

line. The **DBUG\_PROCESS** macro is generally used only once per program to inform the debugger what name the program was invoked with. The **DBUG\_PUSH** macro modifies the current debugger state by saving the previous state and setting a new state based on the control string passed as its argument. The **DBUG\_PRINT** macro is used to print the values of each argument for which a factorial is to be computed. The **DBUG\_RETURN** macro tells the debugger that the end of the current function has been reached and returns a value to the calling function. All of these macros will be fully explained in subsequent sections.

To use the debugger, the factorial program is invoked with a command line of the form:

```
factorial -#d:t 1 2 3
```

The **main** function recognizes the "-#d:t" string as a debugger control string, and passes the debugger arguments ("d:t") to the *debug* runtime support routines via the **DBUG\_PUSH** macro. This particular string enables output from the **DBUG\_PRINT** macro with the 'd' flag and enables function tracing with the 't' flag. The factorial function is then called three times, with the arguments "1", "2", and "3". Note that the **DBUG\_PRINT** takes exactly **two** arguments, with the second argument (a format string and list of printable values) enclosed in parentheses.

Debug control strings consist of a header, the "-#", followed by a colon separated list of debugger arguments. Each debugger argument is a single character flag followed by an optional comma separated list of arguments specific to the given flag. Some examples are:

```
-#d:t:o
-#d,in,out:f,main:F:L
```

Note that previously enabled debugger actions can be disabled by the control string "-#".

The definition of the factorial function, symbolized as "N!", is given by:

$$N! = N * N-1 * ... 2 * 1$$

Figure 5 is the factorial function which implements this algorithm recursively. Note that this is not necessarily the best way to do factorials and error conditions are ignored completely.

```

#ifdef DEBUG_OFF                                /* We are testing dbug */

int factorial(register int value) {
    if(value > 1) {
        value *= factorial(value-1);
    }
    return value;
}

#else

#include <my_global.h>

int factorial (
register int value)
{
    DEBUG_ENTER ("factorial");
    DEBUG_PRINT ("find", ("find %d factorial", value));
    if (value > 1) {
        value *= factorial (value - 1);
    }
    DEBUG_PRINT ("result", ("result is %d", value));
    DEBUG_RETURN (value);
}

#endif

```

Figure 5  
Factorial Function

One advantage (some may not consider it so) to using the *dbug* package is that it strongly encourages fully structured coding with only one entry and one exit point in each function. Multiple exit points, such as early returns to escape a loop, may be used, but each such point requires the use of an appropriate **DEBUG\_RETURN** or **DEBUG\_VOID\_RETURN** macro.

To build the factorial program on a **UNIX** system, compile and link with the command:

```
cc -o factorial main.c factorial.c -ldbug
```

The "-ldbug" argument tells the loader to link in the runtime support modules for the *dbug* package. Executing the factorial program with a command of the form:

```
factorial 1 2 3 4 5
```

generates the output shown in figure 6.

```

1
2
6
24
120

```

Figure 6  
factorial 1 2 3 4 5

Function level tracing is enabled by passing the debugger the 't' flag in the debug control string. Figure 7 is the output resulting from the command "factorial -#t:o 2 3".

```

| >factorial
| | >factorial
| | <factorial
| <factorial
2
| >factorial
| | >factorial
| | | >factorial
| | | <factorial
| | <factorial
| <factorial
6
<main

```

Figure 7  
factorial -#t:o 2 3

Each entry to or return from a function is indicated by '>' for the entry point and '<' for the exit point, connected by vertical bars to allow matching points to be easily found when separated by large distances.

This trace output indicates that there was an initial call to factorial from main (to compute 2!), followed by a single recursive call to factorial to compute 1!. The main program then output the result for 2! and called the factorial function again with the second argument, 3. Factorial called itself recursively to compute 2! and 1!, then returned control to main, which output the value for 3! and exited.

Note that there is no matching entry point "main>" for the return point "<main" because at the time the **DBUG\_ENTER** macro was reached in main, tracing was not enabled yet. It was only after the macro **DBUG\_PUSH** was executing that tracing became enabled. This implies that the argument list should be processed as early as possible since all code preceding the first call to **DBUG\_PUSH** is essentially invisible to **dbug** (this can be worked around by inserting a temporary **DBUG\_PUSH(argv[1])** immediately after

the `DEBUG_ENTER("main")` macro.

One last note, the trace output normally comes out on the standard error. Since the factorial program prints its result on the standard output, there is the possibility of the output on the terminal being scrambled if the two streams are not synchronized. Thus the debugger is told to write its output on the standard output instead, via the 'o' flag character. Note that no 'o' implies the default (standard error), a 'o' with no arguments means standard output, and a 'o' with an argument means used the named file. i.e, "factorial -#t:o,logfile 3 2" would write the trace output in "logfile". Because of **UNIX** implementation details, programs usually run faster when writing to stdout rather than stderr, though this is not a prime consideration in this example.

**USE OF DEBUG\_PRINT MACRO**

The mechanism used to produce "printf" style output is the **DEBUG\_PRINT** macro.

To allow selection of output from specific macros, the first argument to every **DEBUG\_PRINT** macro is a *debug* keyword. When this keyword appears in the argument list of the 'd' flag in a debug control string, as in "-#d,keyword1,keyword2,...:t", output from the corresponding macro is enabled. The default when there is no 'd' flag in the control string is to enable output from all **DEBUG\_PRINT** macros.

Typically, a program will be run once, with no keywords specified, to determine what keywords are significant for the current problem (the keywords are printed in the macro output line). Then the program will be run again, with the desired keywords, to examine only specific areas of interest.

The second argument to a **DEBUG\_PRINT** macro is a standard printf style format string and one or more arguments to print, all enclosed in parentheses so that they collectively become a single macro argument. This is how variable numbers of printf arguments are supported. Also note that no explicit newline is required at the end of the format string. As a matter of style, two or three small **DEBUG\_PRINT** macros are preferable to a single macro with a huge format string. Figure 8 shows the output for default tracing and debug.

```

| args: argv[2] = 3
| >factorial
| | find: find 3 factorial
| | >factorial
| | | find: find 2 factorial
| | | >factorial
| | | | find: find 1 factorial
| | | | result: result is 1
| | | <factorial
| | | result: result is 2
| | | <factorial
| | | result: result is 6
| | <factorial
| 6
| <main

```

Figure 8  
factorial -#d:t:o 3

The output from the **DEBUG\_PRINT** macro is indented to match the trace output for the function in which the macro occurs. When debugging is enabled, but not trace, the output starts at the left margin, without indentation.

To demonstrate selection of specific macros for output, figure 9 shows the result when the factorial program is invoked with the debug control string "-#d,result:o".

```
factorial: result: result is 1
factorial: result: result is 2
factorial: result: result is 6
factorial: result: result is 24
24
```

Figure 9

```
factorial -#d,result:o 4
```

It is sometimes desirable to restrict debugging and trace actions to a specific function or list of functions. This is accomplished with the 'f' flag character in the debug control string. Figure 10 is the output of the factorial program when run with the control string "-#d:f,factorial:F:L:o". The 'F' flag enables printing of the source file name and the 'L' flag enables printing of the source file line number.

```
factorial.c:      18: factorial: find: find 3 factorial
factorial.c:      18: factorial: find: find 2 factorial
factorial.c:      18: factorial: find: find 1 factorial
factorial.c:      22: factorial: result: result is 1
factorial.c:      22: factorial: result: result is 2
factorial.c:      22: factorial: result: result is 6
6
```

Figure 10

```
factorial -#d:f,factorial:F:L:o 3
```

The output in figure 10 shows that the "find" macro is in file "factorial.c" at source line 8 and the "result" macro is in the same file at source line 12.

## SUMMARY OF MACROS

This section summarizes the usage of all currently defined macros in the *dbug* package. The macros definitions are found in the user include file **dbug.h** from the standard include directory.

**DBUG\_ENTER** Used to tell the runtime support module the name of the function being entered. The argument must be of type "pointer to character". The **DBUG\_ENTER** macro must precede all executable lines in the function just entered, and must come after all local declarations. Each **DBUG\_ENTER** macro must have a matching **DBUG\_RETURN** or **DBUG\_VOID\_RETURN** macro at the function exit points. **DBUG\_ENTER** macros used without a matching **DBUG\_RETURN** or **DBUG\_VOID\_RETURN** macro will cause warning messages from the *dbug* package runtime support module.

EX: `DBUG_ENTER ("main");`

**DBUG\_RETURN** Used at each exit point of a function containing a **DBUG\_ENTER** macro at the entry point. The argument is the value to return. Functions which return no value (void) should use the **DBUG\_VOID\_RETURN** macro. It is an error to have a **DBUG\_RETURN** or **DBUG\_VOID\_RETURN** macro in a function which has no matching **DBUG\_ENTER** macro, and the compiler will complain if the macros are actually used (expanded).

EX: `DBUG_RETURN (value);`  
 EX: `DBUG_VOID_RETURN;`

**DBUG\_PROCESS** Used to name the current process being executed. A typical argument for this macro is "argv[0]", though it will be perfectly happy with any other string. In multi-threaded environment threads may have different names.

EX: `DBUG_PROCESS (argv[0]);`

**DBUG\_PUSH** Sets a new debugger state by pushing the current **dbug** state onto an internal stack and setting up the new state using the debug control string passed as the macro argument. The most common usage is to set the state specified by a debug control string retrieved from the argument list. If the control string is *incremental*, the new state is a copy of the old state, modified by the control string.

EX: `DBUG_PUSH ((argv[i][2]));`  
 EX: `DBUG_PUSH ("d:t");`  
 EX: `DBUG_PUSH ("");`

**DBUG\_POP** Restores the previous debugger state by popping the state stack. Attempting to pop more states than pushed will be ignored and no warning will be given. The **DBUG\_POP** macro has no arguments.

EX: `DBUG_POP ();`

**DBUG\_SET** Modifies the current debugger state on top of the stack using the debug control string passed as the macro argument. Unless *incremental* control string is used (see below), it's equivalent to a combination of **DBUG\_POP** and **DBUG\_PUSH**.

```
EX: DEBUG_SET ("d:t");
EX: DEBUG_SET ("+d,info");
EX: DEBUG_SET ("+t:-d");
```

**DEBUG\_FILE** The **DEBUG\_FILE** macro is used to do explicit I/O on the debug output stream. It is used in the same manner as the symbols "stdout" and "stderr" in the standard I/O package.

```
EX: fprintf (DEBUG_FILE, "Doing my own I/O!\n");
```

**DEBUG\_EXECUTE** The **DEBUG\_EXECUTE** macro is used to execute any arbitrary C code. The first argument is the debug keyword, used to trigger execution of the code specified as the second argument. This macro must be used cautiously because, like the **DEBUG\_PRINT** macro, it is automatically selected by default whenever the 'd' flag has no argument list (i.e., a "#d:t" control string).

```
EX: DEBUG_EXECUTE ("status", print_status ());
```

**DEBUG\_EXECUTE\_IF** Works like **DEBUG\_EXECUTE** macro, but the code is **not** executed "by default", if the keyword is not explicitly listed in the 'd' flag. Used to conditionally execute "dangerous" actions, e.g to crash the program testing how recovery works, or to introduce an artificial delay checking for race conditions.

```
EX: DEBUG_EXECUTE_IF ("crashme", abort ());
```

**DEBUG\_EVALUATE** The **DEBUG\_EVALUATE** macro is similar to **DEBUG\_EXECUTE**, but it can be used in the expression context. The first argument is the debug keyword that is used to choose whether the second (keyword is enabled) or the third (keyword is not enabled) argument is evaluated. When **debug** is compiled off, the third argument is evaluated.

```
EX:
    printf("Info-debug is %s",
          DEBUG_EVALUATE ("info", "ON", "OFF"));
```

**DEBUG\_EVALUATE\_IF** Works like **DEBUG\_EVALUATE** macro, but the second argument is **not** evaluated, if the keyword is not explicitly listed in the 'd' flag. Like **DEBUG\_EXECUTE\_IF** this could be used to conditionally execute "dangerous" actions.

```
EX:
    if (prepare_transaction () ||
        DEBUG_EVALUATE ("crashme", (abort (), 0), 0) ||
        commit_transaction () )
```

**DEBUG\_PRINT** Used to do printing via the "fprintf" library function on the current debug stream, **DEBUG\_FILE**. The first argument is a debug keyword, the second is a format string and the corresponding argument list. Note that the format string and argument list are all one macro argument and **must** be enclosed in parentheses.

```
EX: DEBUG_PRINT ("eof", ("end of file found"));
EX: DEBUG_PRINT ("type", ("type is %x", type));
EX: DEBUG_PRINT ("stp", ("%x -> %s", stp, stp -> name));
```

**DEBUG\_DUMP** Used to dump a memory block in hex via the "fprintf" library function on the current debug stream, `DEBUG_FILE`. The first argument is a debug keyword, the second is a pointer to a memory to dump, the third is a number of bytes to dump.

```
EX: DEBUG_DEBUG ("net", packet, len);
```

**DEBUG\_SETJMP** Used in place of the `setjmp()` function to first save the current debugger state and then execute the standard `setjmp` call. This allows to the debugger to restore its state when the `DEBUG_LONGJMP` macro is used to invoke the standard `longjmp()` call. Currently all instances of `DEBUG_SETJMP` must occur within the same function and at the same function nesting level.

```
EX: DEBUG_SETJMP (env);
```

**DEBUG\_LONGJMP** Used in place of the `longjmp()` function to first restore the previous debugger state at the time of the last `DEBUG_SETJMP` and then execute the standard `longjmp()` call. Note that currently all `DEBUG_LONGJMP` macros restore the state at the time of the last `DEBUG_SETJMP`. It would be possible to maintain separate `DEBUG_SETJMP` and `DEBUG_LONGJMP` pairs by having the debugger runtime support module use the first argument to differentiate the pairs.

```
EX: DEBUG_LONGJMP (env, val);
```

**DEBUG\_LOCK\_FILE** Used in multi-threaded environment to lock `DEBUG_FILE` stream. It can be used, for example, in functions that need to write something to a debug stream more than in one `fprintf()` call and want to ensure that no other thread will write something in between.

```
EX:
    DEBUG_LOCK_FILE;
    fprintf (DEBUG_FILE, "a=");
    for (int i=0; i < a_length; i++)
        fprintf (DEBUG_FILE, "0x%03x ", a[i]);
    fprintf (DEBUG_FILE, "];");
    DEBUG_UNLOCK_FILE;
```

**DEBUG\_UNLOCK\_FILE** Unlocks `DEBUG_FILE` stream, that was locked with a `DEBUG_LOCK_FILE`.

**DEBUG\_ASSERT** This macro just does a regular `assert()`. The difference is that it will be disabled by `DEBUG_OFF` together with the *debug* library. So there will be no need to disable asserts separately with `NDEBUG`.

```
EX: DEBUG_ASSERT( a > 0 );
```

**DEBUG\_EXPLAIN** Generates control string corresponding to the current debug state. The macro takes two arguments - a buffer to store the result string into and its length. The macro (which could be used as a function) returns 1 if the control string didn't fit into the buffer and was truncated and 0 otherwise.

```
EX:
    char buf[256];
    DEBUG_EXPLAIN( buf, sizeof(buf) );
```

**DBUG\_SET\_INITIAL****DBUG\_EXPLAIN\_INITIAL**

These two macros are identical to **DBUG\_SET** and **DBUG\_EXPLAIN**, but they operate on the debug state that any new thread starts from. Modifying *initial* value does not affect threads that are already running. Obviously, these macros are only useful in the multi-threaded environment.

## DEBUG CONTROL STRING

The debug control string is used to set the state of the debugger via the **DEBUG\_PUSH** or **DEBUG\_SET** macros. Control string consists of colon separate flags. Colons that are part of `:\`, `:/`, or `::` are not considered flag separators. A flag may take an argument or a list of arguments. If a control string starts from a '+' sign it works *incrementally*, that is, it can modify existing state without overriding it. In such a string every flag may be preceded by a '+' or '-' to enable or disable a corresponding option in the debugger state. This section summarizes the currently available debugger options and the flag characters which enable or disable them. Argument lists enclosed in '[' and ']' are optional.

- a[,file] Redirect the debugger output stream and append it to the specified file. The default output stream is stderr. A null argument list causes output to be redirected to stdout.  
  
EX: a,C:\tmp\log
- A[,file] Like 'a[,file]' but ensure that data are written after each write (this typically implies flush or close/reopen). It helps to get a complete log file in case of crashes. This mode is implicit in multi-threaded environment.
- d[,keywords] Enable output from macros with specified keywords. An empty list of keywords implies that all keywords are selected.
- D[,time] Delay for specified time after each output line, to let output drain. Time is given in tenths of a second (value of 10 is one second). Default is zero.
- f[,functions] Limit debugger actions to the specified list of functions. An empty list of functions implies that all functions are selected.
- F Mark each debugger output line with the name of the source file containing the macro causing the output.
- i Mark each debugger output line with the PID (or thread ID) of the current process.
- g[,functions] Enable profiling for the specified list of functions. An empty list of functions enables profiling for all functions. See **PROFILING WITH DEBUG** below.
- L Mark each debugger output line with the source file line number of the macro causing the output.
- n Mark each debugger output line with the current function nesting depth.
- N Sequentially number each debugger output line starting at 1. This is useful for reference purposes when debugger output is interspersed with program output.
- o[,file] Like 'a[,file]' but overwrite old file, do not append.
- O[,file] Like 'A[,file]' but overwrite old file, do not append.
- p[,processes] Limit debugger actions to the specified processes. An empty list implies all processes. This is useful for processes which run child processes. Note that each debugger output line can be marked with the name of the current process via the 'P' flag. The process name must match the argument passed to the **DEBUG\_PROCESS** macro.
- P Mark each debugger output line with the name of the current process. Most useful when used with a process which runs child processes that are also being debugged. Note that the parent process must arrange for the debugger control string to be passed to the child processes.

- r Used in conjunction with the **DBUG\_PUSH** macro to reset the current indentation level back to zero. Most useful with **DBUG\_PUSH** macros used to temporarily alter the debugger state.
- S When compiled with *safemalloc* this flag forces "sanity" memory checks (for overwrites/underwrites) on each **DBUG\_ENTER** and **DBUG\_RETURN**.
- t[,N] Enable function control flow tracing. The maximum nesting depth is specified by N, and defaults to 200.
- T Mark each debugger output line with the current timestamp. The value is printed with microsecond resolution, as returned by *gettimeofday()* system call. The actual resolution is OS- and hardware-dependent.

## MULTI-THREADED DEBUGGING

When *dbug* is used in a multi-threaded environment there are few differences from a single-threaded case to keep in mind. This section tries to summarize them.

- Every thread has its own stack of debugger states. **DBUG\_PUSH** and **DBUG\_POP** affect only the thread that executed them.
- At the bottom of the stack for all threads there is the common *initial* state. Changes to this state (for example, with **DBUG\_SET\_INITIAL** macro) affect all new threads and all running threads that didn't **DBUG\_PUSH** yet.
- Every thread can have its own name, that can be set with **DBUG\_PROCESS** macro. Thus, "#p,name1,name2" can be used to limit the output to specific threads.
- When printing directly to **DBUG\_FILE** it may be necessary to prevent other threads from writing something between two parts of logically indivisible output. It is done with **DBUG\_LOCK\_FILE** and **DBUG\_UNLOCK\_FILE** macros. See the appropriate section for examples.
- "#o,file" and "#O,file" are treated as "#a,file" and "#A,file" respectively. That is all writes to a file are always followed by a flush.
- "#i" prints not a PID but a thread id in the form of "T@nnn"

**PROFILING WITH DBUG**

With *debug* one can do profiling in a machine independent fashion, without a need for profiled version of system libraries. For this, *debug* can write out a file called **debugmon.out** (by default). This is an ascii file containing lines of the form:

```
<function-name> E <time-entered>
<function-name> X <time-exited>
```

A second program (**analyze**) reads this file, and produces a report on standard output.

Profiling is enabled through the **g** flag. It can take a list of function names for which profiling is enabled. By default, it profiles all functions.

The profile file is opened for appending. This is in order that one can run a program several times, and get the sum total of all the times, etc.

An example of the report generated follows:

```

Profile of Execution
Execution times are in milliseconds

          Calls                               Time
          -----                               ----
Function  Times  Percentage  Time Spent  Percentage  Importance
=====  =====  =====  =====  =====  =====
factorial    5      83.33      30        100.00      8333
main         1      16.67       0         0.00         0
=====  =====  =====  =====  =====  =====
Totals      6      100.00     30        100.00
    
```

As you can see, it's quite self-evident. The **Importance** column is a metric obtained by multiplying the percentage of the calls and the percentage of the time. Functions with higher 'importance' benefit the most from being sped up.

As a limitation - setjmp/longjmp, or child processes, are ignored for the time being. Also, profiling does not work in a multi-threaded environment.

Profiling code is (c) Binayak Banerjee.

## HINTS AND MISCELLANEOUS

One of the most useful capabilities of the *dbug* package is to compare the executions of a given program in two different environments. This is typically done by executing the program in the environment where it behaves properly and saving the debugger output in a reference file. The program is then run with identical inputs in the environment where it misbehaves and the output is again captured in a reference file. The two reference files can then be differentially compared to determine exactly where execution of the two processes diverges.

A related usage is regression testing where the execution of a current version is compared against executions of previous versions. This is most useful when there are only minor changes.

It is not difficult to modify an existing compiler to implement some of the functionality of the *dbug* package automatically, without source code changes to the program being debugged. In fact, such changes were implemented in a version of the Portable C Compiler by the author in less than a day. However, it is strongly encouraged that all newly developed code continue to use the debugger macros for the portability reasons noted earlier. The modified compiler should be used only for testing existing programs.

## CAVEATS

The *dbug* package works best with programs which have "line oriented" output, such as text processors, general purpose utilities, etc. It can be interfaced with screen oriented programs such as visual editors by redefining the appropriate macros to call special functions for displaying the debugger results. Of course, this caveat is not applicable if the debugger output is simply dumped into a file for post-execution examination.

Programs which use memory allocation functions other than **malloc** will usually have problems using the standard *dbug* package. The most common problem is multiply allocated memory.

# **D B U G**

## **C Program Debugging Package**

**by**

*Fred Fish*

### *ABSTRACT*

This document introduces *debug*, a macro based C debugging package which has proven to be a very flexible and useful tool for debugging, testing, and porting C programs.

All of the features of the *debug* package can be enabled or disabled dynamically at execution time. This means that production programs will run normally when debugging is not enabled, and eliminates the need to maintain two separate versions of a program.

Many of the things easily accomplished with conventional debugging tools, such as symbolic debuggers, are difficult or impossible with this package, and vice versa. Thus the *debug* package should *not* be thought of as a replacement or substitute for other debugging tools, but simply as a useful *addition* to the program development and maintenance environment.