Spotless 2 - how to use it

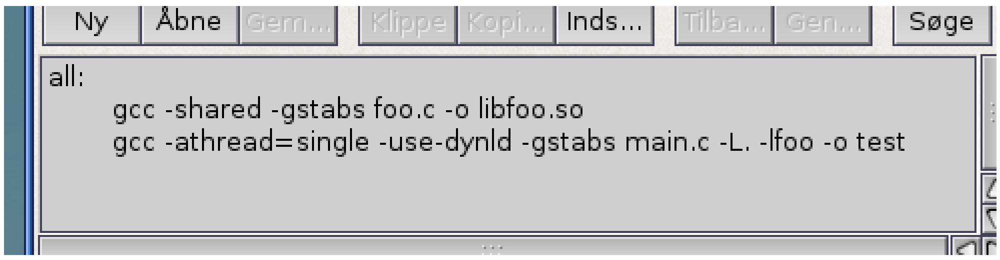
Intro

Spotless is - if you know how to use it - extremely simple to use.

The design phase of its development has taken into account reducing the amount of interaction elements, so only the necessary functions are visible in the ui. For instance, most of the functions have been implemented using the buttons and interactions elements in the window(s), to reduce superfluous use of menus and lists.

First : You need a project

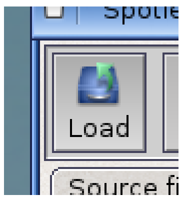
To use a code project with Spotless, you need to know one thing : Spotless is hardcoded to use the stabs format. This fact is more due to the experience of the developer than a question of usefulness. For instance, modern code projects in c and c++ might use precompiled headers, which are not supported by the stabs format. But appart from that, stabs should serve your needs.



```
all:
	gcc -shared -gstabs foo.c -o libfoo.so
	gcc -athread=single -use-dynld -gstabs main.c -L. -lfoo -o test
```

To compile a source file with stabs (Symbol TABleS) symbols, you feed the gcc (or g++) compiler with the -gstabs flag. To link an executable or a shared object with stabs, you also give it the -gstabs flag. To build a static library with stabs, you just need to make sure that the individual object files are a result of a compile command using the -gstabs flag.

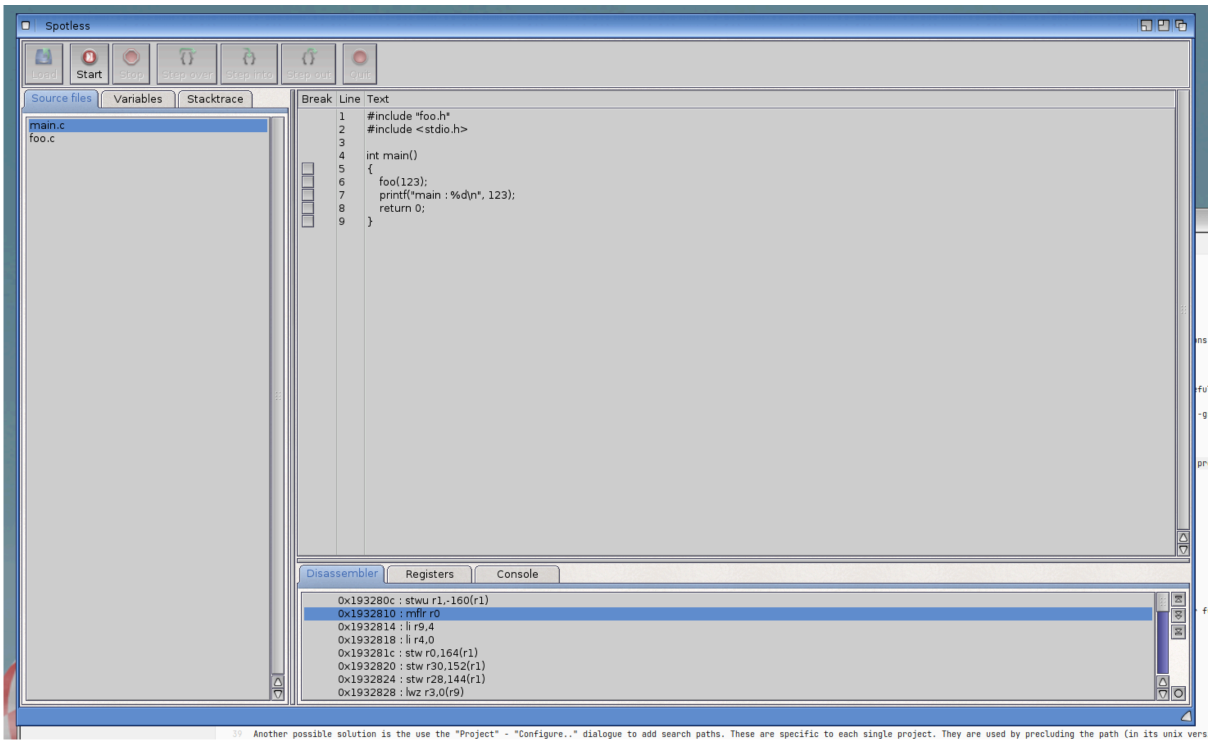
Second : Load it into Spotless



Easy. Use the load button. Upon load, Spotless will ask you for command arguments for the executable. Unless you are debugging a command line tool with inputs, you can just click "Ok". If you prefer not to be asked about command line arguments on each and every load, you can unclick the "Ask for arguments" box in the project menu.

If your project has been properly built (see first chapter), then you should see a list of source file names (perhaps with full paths) in the source file tab to the left of the main window.

Third : Basic Interactions



```
#include "foo.h"
#include <stdio.h>

int main()
{
   foo(123);
   printf("main : %d\n", 123);
   return 0;
}
```

```
0x193280c : stwu r1,-160(r1)
0x1932810 : mflr r0
0x1932814 : li r9,4
0x1932818 : li r4,0
0x193281c : stw r0,164(r1)
0x1932820 : stw r30,152(r1)
0x1932824 : stw r28,144(r1)
0x1932828 : lwz r3,0(r9)
```

```
Another possible solution is the use the "Project" - "Configure.." dialogue to add search paths. These are specific to each single project. They are used by precluding the path (in its unix vers
```

Interaction with Spotless should be pretty intuitive : You click a source file name, find the place in the code you want to inspect, and then you insert breakpoints to interact with it.

But there is quite a big question of how to find the source files.

Finding the source files :

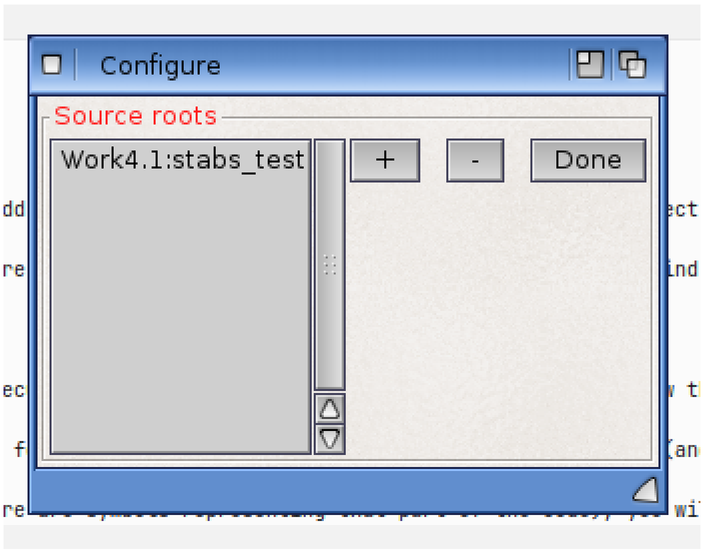
The main window has been built, so that the largest center area is reserved to browse a single source file. To show the source code for a source file entry, you just need to click the name (or full path, depending on the build) in the "Source files" pane to the left. If Spotless is not able to locate the particular source file based on the entry string, it will tell you so in the main area. There are several ways to help Spotless to detect source files in their correct locations. If the path string (the actual string shown in the "Source files" pane) is a full unix path, the you can "trick" Spotless into realizing the path by creating an assign to the root element and copying your entire source folder there. For instance, if your path string says

1. /home/beetlejuice/mysuperapp/source/hello.cpp

... then the obvious solution is to create a folder hierachy on your hd called "home/beetlejuice", copying your "mysuperapp" folder there and assigning

1. assign home: work:home/beetlejuice

Or some similar solution. I am sure you can work it out.

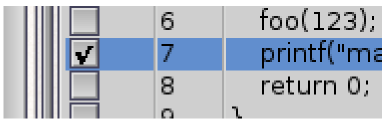


Another possible solution is to use the "Project" - "Configure.." dialogue to add search paths. These are specific to each single project. They are used by precluding the path (in its unix version) to the path shown in the "Source files" pane.

This is probably the most complicated part of the interaction with Spotless. There is no real generic solution to solving the "how-to-find-the-source-files" problem, when creating a debugger - especially not if you are using a cross compiler. The two tools described here should be optimal in terms of solving it.

Using breakpoints to interact with the code

The basic user interaction for a debugger is to allow the user to "break" the execution of a program in a specific place, and then allow the user to inspect and interact with the code in that specific place.



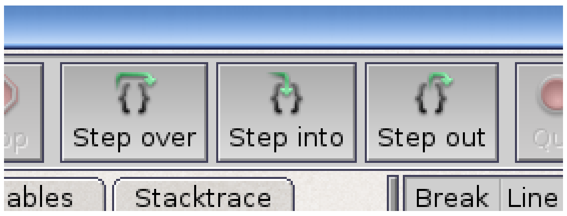
```
6    foo(123);
7    printf("ma
8    return 0;
```

The source code area (the biggest area in the main window) will display the code for a specific source object. If there are code lines (and not just struct and variable defines) in that object, there will be checkboxes on those lines, that represent code, in the "Break" column of the "Source" text area. To activate a breakpoint on a line, simply check the box representing the line you want to break at. When all is set (you might set multiple such breakpoints), then you can start (or continue) program execution by pressing "Start". This will unpause the child. Explanation : When you load your program into Spotless, it actually starts execution of it, just like if it was loaded using Workbench or Shell. But it is set to pause before it executes any instructions. Whenever your program is in a paused state, the "Start" button will be unghosted/selectable. Whenever you program is running, it will be ghosted, so you cannot interact with it this way. Instead, the "Stop" button will be available as ui element. Be aware that if you use the "Stop" button at any given time, you might not get into an ideal situation. Your program may call library functions that are not part of the source base. If the instruction that you pause on (the instruction about to be executed by the program when you press "Stop") doesn't have corresponding symbols (stabs), then you will not be able to read source line information, variables and so on. You will, however, still be able to read disassembly, registers, stack trace, and you will be able to restart the program.

When your program is in a paused state, AND you have context information (if there are symbols representing that part of the code), you will be able to a) read variables in the "Variables" pane and b) do stepwise execution.

Stepwise execution

The buttons "Step over", "Step into", "Step out" represent different ways to do "stepwise execution".



"Step over" is the legacy name for an operation that executes the present (current) line in the code and then immediately stops at the line coming just after. The term is perhaps a tad misleading, since the actual operation does not really step "over", but rather "through". The actual term for stepping "over" (and NOT executing the intermediate code) is "skip". This operation is not available in the main window of Spotless, but it can be found in the window called the "Memory Surfer". This is a secret window, and unless you know what you are playing with, you should not use it. However, for those functions that require full control, it is useful. It can be found by clicking the little 'o' button in the lowest right corner of the "Disassembler" pane and in the "Windows" menu.

"Step into" also executes the present code bit, but it has the difference from "Step over", that IF the current code line involves a subroutine call (for instance if it calls an object method or another function (or the same function in cases of self-reference :)), then the execution will stop at the first instruction in that particular sub-routine. When stopping, it will (if possible) show the source line of that sub-routine, and if this is contained in another source file, this new source file will be loaded.

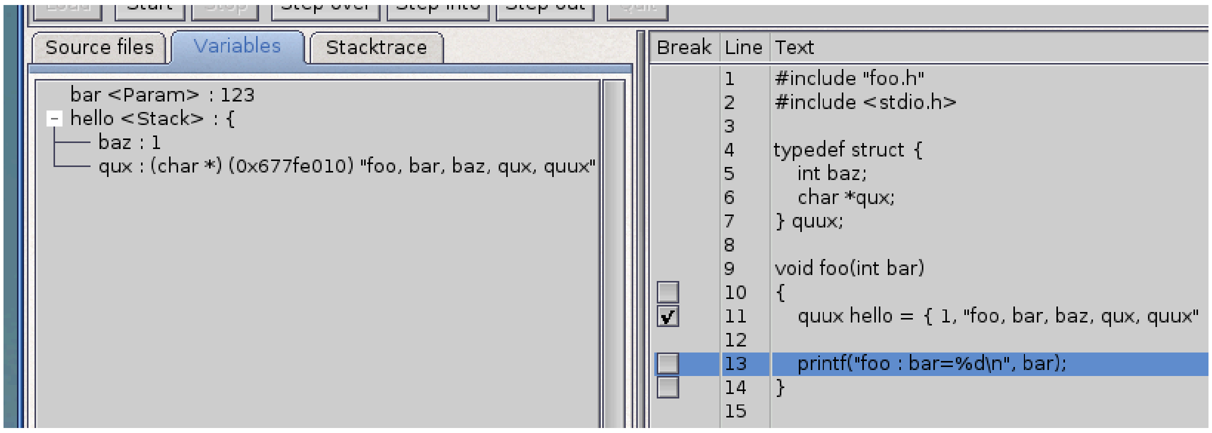
"Step out" is a strange function. It also steps, but this time the execution is not stopped on the next line but only when the program returns from the current subroutine. This means that if the current subroutine has been called from, say, the main() function, then Spotless will detect any "return" statement (either explicit or implicit) and stop execution at the instruction immediately after the return instruction. IF (and only if) the originating function contains stabs (if it has a symbolic representation in the debug information of the executable), then it will be displayed.

On the subject of optimization : Due to the character of optimizations used by the compiler, the order of execution of source files are not always the same as the order of the source lines in the source file (text file). Furthermore, the same line might be represented more than once, and this can be both just after or after other source lines. Due to these facts, it is not possible to be 100% logical in the way stepwise execution is done when using Spoless on executables that have been built using some kind of optimization. The only way to circumvent this is to disable all optimization. And even then, it is not absolutely granted, that you get and immediate sequence.

On the subject of inline functions : Same as above, but take into account that now your stacktrace will also be - to use a selective wording - "funky".

Variables

Spotless has been developed to take care of many types of variables. It will correctly detect many symbol types and correctly display values, if they are available. When all that is said and done, this is by far the most difficult part of building a debugger - that the ways compilers represent symbols are magnificent.



```
bar <Param> : 123
hello <Stack> : {
    baz : 1
    qux : (char *) (0x677fe010) "foo, bar, baz, qux, quux"
```

```
#include "foo.h"
#include <stdio.h>

typedef struct {
   int baz;
   char *qux;
} quux;

void foo(int bar)
{
   quux hello = { 1, "foo, bar, baz, qux, quux"

   printf("foo : bar=%d\n", bar);
}
```

First : A simple variable might refer to an unreadable address. Spotless will check the memory status of the pointer and write if the address would give a crash on read. NOTE : This particular problem of reading any address is maybe not 100 % tackled. There might, to be specific, be readability issues on some platforms.

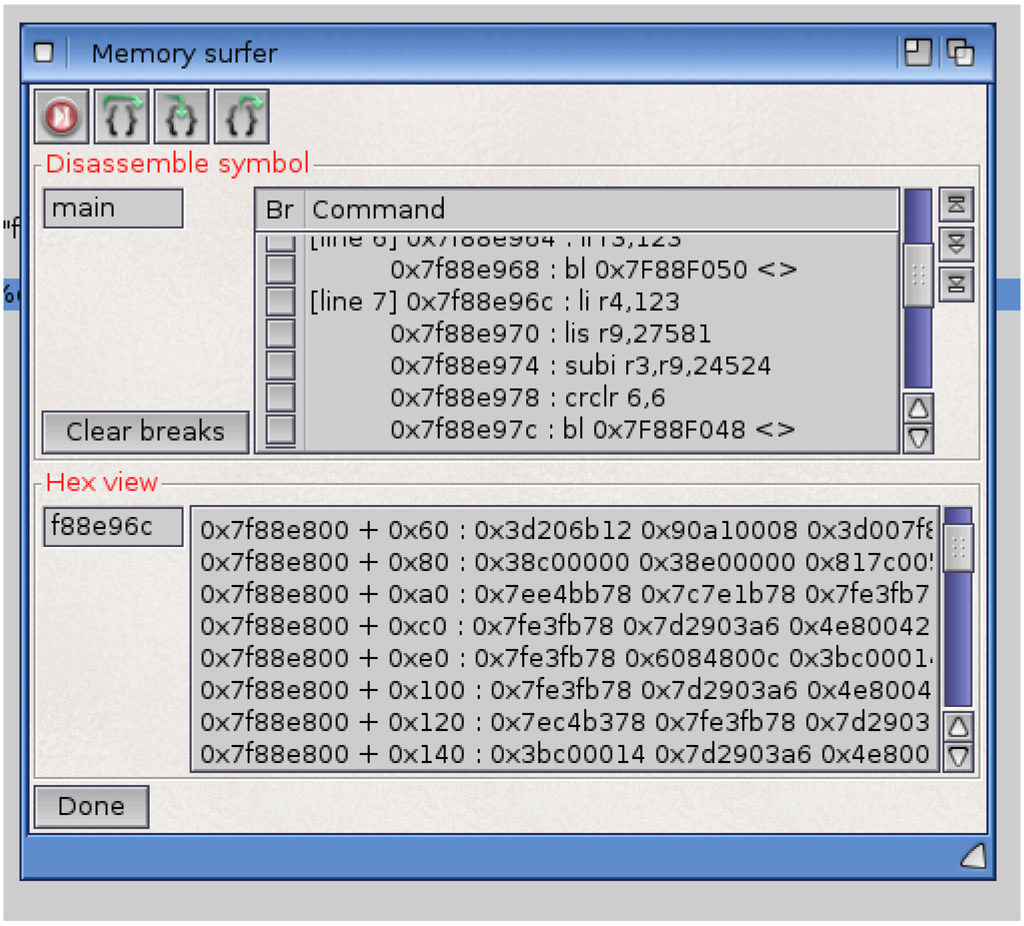
Second : Self referencing lists and structs. If you give Spotless a function with a struct, that has some kind of self-reference (for instance some lists and arrays), then there is a maximum on the amount of sublevels, that Spotless will read (obviously - otherwise it would try to read infinite levels of that struct) a maximum of sub-levels. This has influence on reads that have lots of sub-levels, because it will not allow you to read them all.

Third : Strings. Spotless will not read unicode strings. Regular strings are not 100 % supported due to the fact that they have so many, many variable types as generators.

Fourth : Format of variable write outs. The format, in which variables are output, is a bit strange in some situations. For instance arrays are not necessarily very comfortable. Improvements are prone to exist.

Fourth : The secrets of the Memory Surfer

There is a "hidden" window in Spotless called the Memory Surfer. Try opening it from the "Windows" menu.



```
[line 6] 0x7f88e964 : li r3,123
         0x7f88e968 : bl 0x7F88F050 <>
[line 7] 0x7f88e96c : li r4,123
         0x7f88e970 : lis r9,27581
         0x7f88e974 : subi r3,r9,24524
         0x7f88e978 : crclr 6,6
         0x7f88e97c : bl 0x7F88F048 <>
```

```
0x7f88e800 + 0x60 : 0x3d206b12 0x90a10008 0x3d007f8
0x7f88e800 + 0x80 : 0x38c00000 0x38e00000 0x817c00
0x7f88e800 + 0xa0 : 0x7ee4bb78 0x7c7e1b78 0x7fe3fb7
0x7f88e800 + 0xc0 : 0x7fe3fb78 0x7d2903a6 0x4e80042
0x7f88e800 + 0xe0 : 0x7fe3fb78 0x6084800c 0x3bc0001
0x7f88e800 + 0x100 : 0x7fe3fb78 0x7d2903a6 0x4e8004
0x7f88e800 + 0x120 : 0x7ec4b378 0x7fe3fb78 0x7d2903
0x7f88e800 + 0x140 : 0x3bc00014 0x7d2903a6 0x4e800
```

This is a lowlevel assembly debugger. As you can see, there are the three stepwise execution buttons. They work the same as the ones in the main window except that they do steps on assembly level. So for instance "Step over" will just execute the next cpu instruction.

In the "Disassemble symbol" box, you can disassemble any symbol, that Spotless recognize in the loaded executable (and shared objects). You should be careful here, because this means, that you can also "disassemble" a variable, which of course doesn't make any sense. But that is the kind of price you pay for power. When you disassemble a function/subroutine, then the instructions are shown in the listbrowser. Note that now you have a breakpoint checkbox for each and every instruction. If you end up with too many breakpoints, you can clear them all (in this function) with the "Clear breaks" button. The small, mysterious buttons to the right of the listbrowser is 1) skip backwards 2) execute instruction (same as assembly "Step over") and 3) skip forwards.

In the "Hex view" box you have the possibility to view the hex dump of any given memory address. When disassembling a symbol, the hex viewer will also show you the hex dump of that entire function.

Fifth : An example

1) Download and unpack this : Test project

2) Open a Shell, cd into the folder, you just unpacked.

3) run "make" to create an executable called "test".

4) Open Spotless and open "test".

5) Take a look around. This is a minimal project, so only two source files to inspect. Because the program is paused before the first instruction, there is not much else to show.

6) Open "foo.c". Look at the code and make sure, that you more or less understand it.

7) Click the checkbox to the left of line 11.

8) Click "Start". Spotless should now highlight line 11 to show, that it has stopped at this point.

9) Click the "Variables" pane to the left. You should now see some variable writeouts. The variable "bar" has value 123 (int). The struct "hello" has not been initialized yet, so the values of the struct members are meaningless.

10) Click "Step over" once. You should now see, that the members of "hello" are now initialized to the values described in the code.

11) Fantastic! You now understand something. Click "Start". The "test" program will exit, and Spotless will return to the initial state, where you can load a new program.

Note :

As you might have noticed, once you completed this exercise, the "test" program uses a shared object. This is now fully supported in Spotless. However, there is no support for loading regular amigaos libraries (*.library). If your program uses these, you will not be able to debug code inside them.

Download Spotless 2 : Spotless 2 on os4depot.net