BlackBox Debugger

Reference Manual

Table of Contents

What is BlackBox?

BlackBox is a command-line, source-level debugger for debugging programs compiled with Catalina.

Black***Box*** should not be confused with Black***Cat***. Black*Cat* was a Windows application with a sophisticated graphical user interface. Black*Box* is a simple command-line debugger that is supported on both Windows and Linux. However, BlackBox and BlackCat are intentionally very similar in style and functionality, so that users of BlackCat (no longer supported and now deprecated) will have no problems switching to BlackBox.

Who wrote it?

BlackBox was written from scratch by Ross Higson, but owes much credit to Bob Anderson's original design for BlackCat. It makes use of the same BlackCat debug cog and debug information file generator initially developed by Bob for BlackCat.

Status

The current release of BlackBox can be used with Catalina Release 4.0 or later.

Features

- A true source level-debugger for C programs;
- Supports both the Propeller 1 and the Propeller 2;
- Display source, set and clear breakpoints, examine and set variables, read and write RAM locations, and display program stack traces;
- Single stepping and user breakpoints supported.
- Minimal debugging overhead – one cog and (approximately) 300 bytes of initialization code.
- Programs execute at full speed except when stopped at a breakpoint, making BlackBox suitable for use with real-time programs;
- Supports debugging of LMM, CMM and XMM programs;
- Supports on all platforms on which Catalina is supported;
- Runs under Linux and Windows;
- Free!

Limitations

- BlackBox cannot debug XMM programs executing from SPI FLASH (e.g. on the C3). To debug these programs, recompile them to use SPI RAM (e.g. use **–x2** instead of –**x4**, or **–x5** instead of **–x3**).
- Catalina cannot debug multi-threaded C programs.

Using BlackBox

Preparing your Propeller platform

BlackBox requires a serial connection with the loaded program.

On Propeller 1 platforms this can be either use the normal serial connection also used to load programs (e.g. a PropPlug connected to pins 30 and 31), or the special serial cable supported by Catalina **payload**, which can be connected to a mouse or keyboard port on the Propeller board.

The latter is useful if the program being debugged itself uses the normal serial port, and is ***necessary*** when debugging XMM programs on Propeller 1 platforms that use the HX512 SRAM card (e.g. the Hydra and Hybrid) since the normal serial port ***cannot be used*** while the HX512 SRAM card is in use. Refer to the **Catalina Reference Manual** for more details, including a schematic of the necessary cable.

On other Propeller 1 platforms, BlackBox by default uses pins 30 & 31. if you need to debug a program that uses a serial HMI on those pins, you will need to have a Prop Plug connected to some other pins, and then modify the pins specified in the platform configuration file, which by default are defined as follows:

```
SI_PIN      = 31
SO_PIN      = 30
...
BLACKCAT_RXPIN = SI_PIN
BLACKCAT_TXPIN = SO_PIN
```

Note that you should change the **BLACKCAT** pin definitions, not the **SI** and **SO** pin definitions, which are also used for other purposes.

On Propeller 2 platforms, BlackBox by default uses pin 62 & 63. which are generally also used for loading programs and for serial HMI options. if you need to debug a program that uses a serial HMI on those pins, you will need to have a Prop Plug connected to some other pins, and then modify the pins specified in the platform configuration file, which by default are defined as follows:

```
#define _BLACKCAT_RX_PIN 63
#define _BLACKCAT_TX_PIN 62
```

Preparing your program

To debug a program with BlackBox, it must be compiled using the Catalina compiler, with either the **-g** or **–g3** command line option specified. These options tell Catalina to add the debug cog to the program (which must have a free cog available to accommodate it), add some initialization code and generate a debugging information file and a listing in addition to the normal program binary file. Catalina provides two levels of debugging support: **–g** and **–g3.**

Either option can be used with BlackBox, although **–g3** is recommended[1]:

- The **–g** option makes minimal changes to the compiled program, and is recommended if space is at a premium, or execution speed is critical.
- The **–g3** option (note that unlike most Catalina options, there must be no space between the **g** and the **3**) allows better debugging capabilities (as described in the **Trace** command and **Step Out** commands, below) but does result in some changes to the compiled C program. These can make the program slightly larger, and make it execute a little more slowly. This will only be a problem if your program already requires all available RAM, or is having trouble executing fast enough.

Loading your program

The binary file of the program to be debugged is loaded into the Propeller as normal (e.g. by using the Catalina Payload program). Programs that use XMM RAM require the loader utilities be built using the **build_utilities** script. See The **Catalina Reference Manual** for more details on both **payload** and the **build_utilities** script, and the **Putting it all together** section (below) for some actual examples.

When a program compiled with the **–g** or **–g3** option is loaded, all device drivers and plugins will be loaded and started, but the program's C code will not begin to execute (at the **main** function) until instructed to do so by BlackBox.

[1] **-g** and **-g3** are used because **lcc** (on which Catalina is based) already assigns specific meanings to **-g1** and **-g2**

Starting BlackBox

After the binary program is loaded into the Propeller, BlackBox can be started on the PC specifying the name of the program **.dbg** file.

The **.dbg** file is generated by Catalina when the **–g** or **–g3** command line option is specified). This file contains debug symbol and program information required to allow BlackBox to interact correctly with the program running on the Propeller.

The BlackBox command line format is:

```
blackbox [ options ] program_name[.dbg]
```

The options are:

```
-? or -h  print a help message and exit (-v -h for more help)
-a port   find propeller port automatically, starting at port
-b baud   baud rate to use (default is 115200)
-d        diagnostic mode (-d again for even more diagnostics)
-L name   execute the named Lua script after opening the port
-p port   port to use
-t msec   timeout to use when opening port (default is 500 msec)
-v        verbose mode (and include port numbers in help message)
-x model  memory model (0 or tiny, 2 or small, 5 or large)
```

If the .dbg extension is not included, it is added automatically by BlackBox.

Blackbox will automatically detect whether a Propeller 1 or Propeller 2 is in use.

If the port is not specified on the command line, BlackBox attempts to find it automatically. Otherwise the port (and baud rate) can be specified using the **Mode** command once BlackBox has started.

To see a list of port numbers that can be used (they will be different depending on whether you are running under Linux or Windows), use the **–v –h** command line options, or the BlackBox **Mode** command from within BlackBox.

BlackBox can automatically detect the memory model used by a loaded program. So the **–x** parameter is not generally required. If specified, it must match the setting of the **–x** command line option used when the program was compiled. It can also be specified using the **Xmm** command once BlackBox has started.

On startup, BlackBox automatically locates the C program's **main** function, and sets the current source line to the first line of that function.

Putting it all together

A complete set of commands to compile, load and debug *hello_world.c* using a Propeller 1 C3 board and using a VGA HMI option might be[2]:

```
set CATALINA_DEFINE=C3 VGA NO_MOUSE
catalina -lc hello_world.c -g3
payload hello_world
blackbox hello_world
```

[2] On Linux, you would use **export** in place of **set**

A complete set of commands to compile, load and debug *hello_world.c* as an XMM program using a Propeller 2 EDGE board and using a VGA HMI option might be:

```
set CATALINA_DEFINE=P2_EDGE LARGE VGA NO_MOUSE
catalina -p2 -lc hello_world.c -g3
payload xmm hello_world
blackbox hello_world
```

Note the inclusion of the **xmm** loader in the **payload** command above. This requires that the **build_utilities** batch file has been executed. This is an interactive batch file – to execute it, simply enter the following command and then follow the prompts:

```
build_utilities
```

BlackBox commands and parameters

All BlackBox commands are entered at the **>** prompt that appears when BlackBox is started. Commands consist of one or more keywords, and each keyword can be abbreviated to any unambiguous length (usually, down to a single character). Parameters follow the keywords.

A space must appear between each command keyword, and between each command keyword and parameter.

For example, the **Step Next** command could be entered as any of:

```
step next
step n
s next
s n
```

(etc)

Numbers can be entered in decimal or hexadecimal format (using the prefix **0x** or **$** to indicate hexadecimal). Also, float, long, short (or word) and char (or byte) values can all be specified. Characters can be entered as character literals (e.g. **'c'**), or as decimal or hexadecimal numbers. If no type is specified, the value is assumed to be a long.

For example:

```
0xABCDABCD
$ABCDABCD
100
-3
long 0xABCDEF
short $ABCD
word -9999
long 4000000
byte 0xFE
char 'c'
byte '\n'
float 3.72
float -1.0e23
```

In addition, the following standard C character literals are supported for bytes or chars:

```
'\\'
'\''
'\0'
'\b'
'\b'
'\f'
'\n'
'\r'
'\t'
'\v'
```

Memory locations can be specified as Cog RAM, Hub RAM or XMM RAM. Cog RAM locations that represent special Catalina registers can be identified by name. If no specific type of RAM location is specified, it is assumed to be a Hub RAM location.

For example:

```
PC
SP
FP
RI
BC
BA
BZ
CS
R0 .. R23
cog 123
co $1f0
c 0x1A
hub $07FFF
hu 0x10
h 0x0000
xmm $ABCDEF
xm 0xFEDCBA
x 0
0x7fff
$10
```

Listing formats

When lines are listed by BlackBox, the format is as follows:

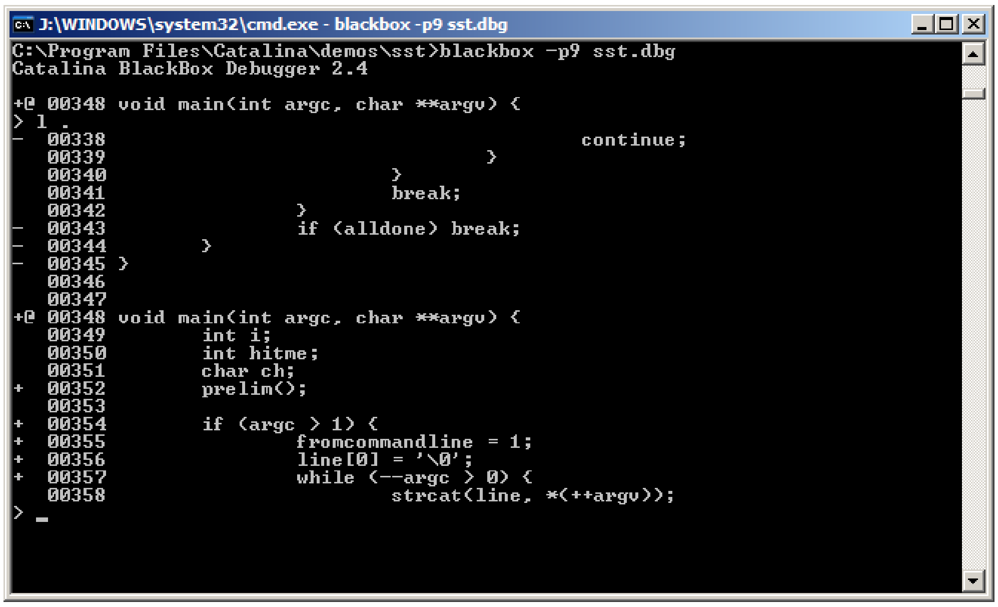
```
XY DDDDD <text of line>
```

Where:

```
X = - if breakpoints can be placed on this line
```

```
    + if a virtual breakpoint has been placed on this line
    * if a user breakpoint has been placed on this line
      or blank otherwise
Y = @ if the program is currently stopped at this line
      or blank otherwise
DDDDD = the line number.
```

For example, here is what BlackBox looks like after starting the program and listing the current debugger location using the command **l .** (see the **List** command for details):



```
C:\Program Files\Catalina\demos\sst>blackbox -p9 sst.dbg
Catalina BlackBox Debugger 2.4

+@ 00348 void main(int argc, char **argv) {
> l .
-  00338                                        continue;
   00339                                }
   00340                        }
   00341                        break;
   00342                }
-  00343                if (alldone) break;
-  00344        }
-  00345 }
   00346
   00347
+@ 00348 void main(int argc, char **argv) {
   00349        int i;
   00350        int hitme;
   00351        char ch;
+  00352        prelim();
   00353
+  00354        if (argc > 1) {
+  00355                fromcommandline = 1;
+  00356                line[0] = '\0';
+  00357                while (--argc > 0) {
   00358                        strcat(line, *(++argv));
> _
```

The above listing shows source lines 338 to 358. The program is currently stopped at the entry to the **main** function (line 348). This is indicated by the **@** character in the second column. That line, along with lines 352, 354 and the other lines with **+** in the first column have virtual breakpoints currently set. Lines 338, 343, and the other lines with **–** in the first column *could* have virtual breakpoints set (i.e. there has been code generated for those lines), but currently *do not* have them. Lines 339, 340, and the other lines with a blank prefix have no code associated with them, and cannot have virtual breakpoints set – the program will never stop on those lines.

User and Virtual breakpoints

The BlackBox debugger supports two types of breakpoints – user and virtual. Each type results in a modification to the program itself (to insert a breakpoint instruction into the code being executed), but virtual breakpoints are managed "behind the scenes" by the debugger, whereas user breakpoints are explicitly added or removed by the user.

User breakpoints are set and cleared on single lines by the user (using the **Breakpoint**, **Delete** commands described below). The **Step Up** and **Step External** commands may also delete user breakpoints from the current function.

Virtual breakpoints are set by the Debugger itself, usually on multiple lines, and usually in response to commands where it may not be obvious a breakpoint is being inserted (for more details on the use of virtual breakpoints, see the **Next**, **Into** and **Out** commands described below).

The main advantage of the virtual breakpoint technique is that it allows programs to run at full speed (when not actually stopped at a breakpoint) - an essential capability for debugging real-time programs. It also means that (except as described in this document) programs to be debugged are exactly the same as programs run normally. This is in contrast to many debuggers, where enabling debugging can make the program many times its original size, with different code executed. This can result in bugs (such as race conditions) that appear in the released program, but disappear when debugging is enabled!

The main disadvantage of the virtual breakpoint technique is that it can be confusing when both types of breakpoint are used. For example, if a user breakpoint is used to get a program to stop in a particular function, then the **Next** command may cause the debugger to lose control of the program if the next line executed is a return to the calling function. This is because the **Next** command relies on virtual breakpoints that would normally have been previously inserted by using the **Step In** command to get into the function in the first place. The **Step Up** and **Step External** commands can be used to exit from the function in such cases, but these commands must be used before the function returns. If in doubt, the best course to avoid losing program control is to insert a user breakpoint that will catch the program if there are no virtual breakpoints to do so.

Where am I?

Most BlackBox commands will only work when the program is stopped at a breakpoint, and the operation of many commands depends on identifying where the program is stopped (e.g. the **Print** and **Update** commands will only show variables visible from the current program line).

BlackBox does this automatically – but how do *you* know where the program is stopped? BlackBox offers several methods:

1. The current line is printed whenever the program stops at a breakpoint.
2. The **Files** command can be used. It shows not only the list of file numbers, but the current source location (used when listing) and debug location. For example, it might show:

```
 1 : C:\Program Files (x86)\Catalina\demos\sst\sst.c
 2 : C:\Program Files (x86)\Catalina\demos\sst\ai.c
 3 : C:\Program Files (x86)\Catalina\demos\sst\battle.c
 4 : C:\Program Files (x86)\Catalina\demos\sst\catalina.c
 5 : C:\Program Files (x86)\Catalina\demos\sst\events.c
 6 : C:\Program Files (x86)\Catalina\demos\sst\finish.c
 7 : C:\Program Files (x86)\Catalina\demos\sst\moving.c
 8 : C:\Program Files (x86)\Catalina\demos\sst\planets.c
 9 : C:\Program Files (x86)\Catalina\demos\sst\reports.c
10 : C:\Program Files (x86)\Catalina\demos\sst\setup.c
```

```
list file = 1
list line = 348

debug file = 1 : C:\Program Files (x86)\Catalina\demos\sst\sst.c
debug line = 348
```

3. The **File** or **List** commands can also be used with the special parameter '**.**' which means 'current debug location'. For example **List .** displays 20 lines centered on the current debug location. This line is always identified in the listing by being prefixed with '**@**' (see the example shown in the **Listing Format** section above).
4. Use the **Trace** command (see the section **Reading Stack Traces** below).

BlackBox commands that accept source line numbers (e.g. **List**, **Breakpoint**, **Delete**) always refer to the current *source* file. The current *source* location (i.e. the source file and line) is not always the same as the current *debug* location. To identify or change the current source location (the debug location is always set automatically by BlackBox) use the **File** command.

Reading Stack Traces

BlackBox can print a stack trace whenever the program is stopped at a breakpoint (see the **Trace** command).

The format of stack traces is similar to the listing format except that each line is prefixed by the file number it belongs to (since each line of the trace may be from a different file). The top line of the trace is always the line at which the program is currently stopped. Each line below that shows the line that called the current function.

Note that only functions with stack frames are shown in the trace. If the **–g** option was used to compile the program (instead of the **–g3** option), Catalina may 'optimize away' stack frames from functions that do not need them (e.g. functions with no local variables and function arguments that cannot be passed in registers). These functions are not shown on the stack trace.

For example, a stack frame trace when the program is currently stopped at line 10 in file 3 might look like:

```
file 3 -@ 00010 printf("BlackBox Version 2.4\n");
file 4 -  00034 print_banner_string() {
file 2 *  01001    if (argc <= 1) print_banner();
```

Command Reference

Breakpoint command

This command is used to show all current user breakpoints, or to add a breakpoint to a particular line or function. To add a breakpoint to a line in a file other than the current source file, first use the **File** command.

The syntax is:

```
breakpoint [ function_name | line ]
```

For example:

To print all current user breakpoints:

```
breakpoint
```

or

```
b
```

To add a breakpoint to line 35 in the current file:

```
breakpoint 35
```

or

```
b 35
```

To add a breakpoint to a function named *my_function*:

```
breakpoint my_function
```

or

```
b my_function
```

When a breakpoint is set successfully, the line will be displayed, and the current source line will be set to the displayed line.

Continue command

This command is used to remove all virtual breakpoints and then continue the program. The effect is that the program will only stop at the next user breakpoint.

The syntax is:

```
continue
```

For example:

```
continue
```

or

```
c
```

Delete command

This command is used to delete a user breakpoint, or all user and all virtual breakpoints (it is not possible to delete a specific virtual breakpoint). To delete a breakpoint from a line in a file other than the current source file, first use the **File** command.

The syntax is:

```
delete [ line | all ]
```

Note that to prevent it being executed accidentally, the **all** parameter cannot be abbreviated.

For example:

To delete a breakpoint on line 35 in the current file:

```
delete 35
```

or

```
d 35
```

To delete all user and virtual breakpoints:

```
delete all
```

or

```
d all
```

Exit command

Terminate the debugger, resuming normal program execution (if possible). You will be asked to confirm this action.

The syntax is:

```
exit
```

For example:

```
exit
```

or

```
e
```

File command

Display a list of current source files, or show the current source file and line, or specify a new source file and line. The current source file is used when listing, and when adding or removing breakpoints. The current line is used only when listing. Note that setting the current source file and line does not affect the line the debugger is currently stopped at.

The syntax is

```
file .
```

or

```
file [ filenum [ linenum ] ]
```

For example:

To set the current source file and line to the current debug location:

```
file .
```

or

```
f .
```

To display a list of the source files that make up the current program:

```
file
```

or

```
f
```

To set the current file to the 3rd entry in the file list:

```
file 3
```

or

```
f 3
```

To set the current file to the 3rd entry and also set the current line to the 15th line:

```
file 3 15
```

or

```
f 3 15
```

Help command

This command is used to display a summary of all debugger commands.

The syntax is:

```
help
```

or

```
?
```

For example:

```
help
```

or

```
h
```

or

```
?
```

Into (or Step Into) command

These commands are identical. They request the debugger to continue from the current line, 'stepping into' any function calls. If there are no function calls in the current line, the debugger simply steps to the next source line that contains executable code.

This command relies on virtual breakpoints. Before continuing the program, the debugger puts a virtual breakpoint on the first line in every function in the program. This means that this command can take some time to execute if used in a large program.

The syntax is:

```
into
```

or

```
step into
```

For example:

```
into
```

or

```
step into
```

or

```
step in
```

or

```
i
```

or

```
s i
```

List command

This command can be used to list specified source files, list a specified function, or list the line that the debugger is currently stopped at. If no parameters are specified, the listing continues from the current source line. The current source line is always updated to the last line listed. Lists can be of a single line, a page (20 lines) or a specified selection of lines.

To list lines in a file other than the current source file, first use the **File** command (this is not required when listing by using a function name).

The syntax is:

```
list .
```

or

```
list function_name
```

or

```
list [ line [ [-] line] ]
```

For example:

To list a page of lines centered (if possible) on the line at which the debugger is currently stopped:

```
list .
```

or

```
l .
```

To list a page of lines centered on line 50 of the current file:

```
list 50
```

or

```
l 50
```

To list a page of lines following the current line:

```
list
```

or

```
l
```

To list the first 20 lines of the function named *my_function*:

```
list my_function
```

or

```
l my_function
```

To list lines 20 to 80 of the current file:

```
list 20-80
```

or

```
list 20 80
```

or

```
l 20 80
```

Mode command

This command can be used to view the memory model and other memory-related information, plus the port or baud rate used when communicating with the program being debugged. This command can also be used to set the port and baud rate (the **Xmm** command is used to set the model). These parameters may also be set on the command line during startup (**-p** for port, **-b** for baud rate, **-x** for memory model).

The baud rate specified (the default if not specified is 115200) must match the baud rate specified for the debug cog when the program being debugged was compiled.

If no parameters are specified, the current value of the various parameters is displayed.

The syntax is:

```
mode [ port [ baud ] ]
```

For example:

To set the port to 9 and the baud rate to 11500:

```
mode 9 115200
```

or

```
m 9 19200
```

To display the settings of various parameters:

```
mode
```

or

```
m
```

Next (or Step Next) command

These commands are identical. They request the debugger to continue from the current line to the next line (containing executable code) in the current function. The debugger does not 'step into' (or stop in) any function calls on the current line unless there are user or virtual breakpoints in those functions. If there are not, the debugger simply steps to the next source line that contains executable code.

This command relies on virtual breakpoints. Before continuing the program, the debugger removes any virtual breakpoints it has put on the first line of each function in the program (see the **Out** command), and adds a breakpoint on every line in the current function. This means that this command can take some time to execute if used in a large program.

Also note that compiler optimizations mean that the next statement with code is not always the next statement in the statement sequence. The Catalina compiler can reorganize and eliminate code (this is especially true for while statements) which means that sometimes you cannot put a breakpoint where you might expect to be able to, and also sometimes **Next** will take you to an unexpected line (e.g. when stepping into a while loop, **Next** may take you to the *end* of the loop before executing the first statement in the loop – this can be disconcerting the first time you encounter it).

Note that **Next** from the last line of a function can cause the debugger to lose control of the program unless there are user or virtual breakpoints set in the program (e.g. **Into** was used to enter the current function).

The syntax is:

```
next
```

or

```
step next
```

For example:

```
next
```

or

```
step next
```

or

```
step n
```

or

```
n
```

or

```
s n
```

Out (or Step Out) command

These commands are identical. They request the debugger to step to the next line outside the current function. The debugger does not stop on any line in the current function unless there are user breakpoints on those lines. If there are not, the debugger simply steps to the next source line that contains executable code which is not in the current function.

This command relies on virtual breakpoints. Before continuing the program, the debugger removes any virtual breakpoints it has put on the lines in the current function, and adds breakpoints to every line in the program outside the current function. This means that this command can take some time to execute if used in a large program.

Note that **Out** from a function that subsequently calls other functions may not immediately return to the function from which the current function was called - the program will stop in *any* line outside the current function. For example, if function A calls function B, and function B calls function C, then 'out' from function B may first be to function C. Then 'out' from function C will be back to function B. Only *then* will 'out' from B go back to function A. For an alternative that does not suffer from this problem (but has other limitations) see the **Step Up** command.

The syntax is:

```
out
```

or

```
step out
```

For example:

```
out
```

or

```
step out
```

or

```
o
```

or

```
step o
```

or

```
s o
```

Print command

The print command prints the address of, the value of, or the location pointed to by, the named variable.

The output of this command is affected by the setting of the verbose flag. See the Verbose command. If verbose is off, only the value is printed. If verbose is on, the data type and memory location are also printed.

The command knows how to print structures and arrays. Arrays of chars are printed as strings for readability. Either constants or simple integer program variables can be used as array indexes – BlackBox will fetch the current value of the variable when determining the address of the element of the array to print.

Note that only variables can be printed, and only those in scope of the line at which the debugger is currently stopped. To print the value of arbitrary memory locations instead, see the **Read** command.

The syntax is:

```
print [ & | * ] variable_name
```

Note that only a single address operator (**&**) can be specified, but multiple dereferencing operators (*****) can be specified (e.g. if the variable holds "a pointer to a pointer to a pointer to …" etc).

For example:

To print the value of the 3rd element of the array *dim* in the structure *coord*:

```
print coord.dim[3]
```

or

```
p coord.dim[3]
```

To print just the address:

```
p &coord.dim[3]
```

To print what this points to (assuming the value is a pointer):

```
p *coord.dim[3]
```

To print the ***i***th element of the *dim* array, where ***i*** is an integer variable in the program being debugged:

```
p coord.dim[i]
```

To print the whole *coord* structure:

```
p coord
```

Read command

This command can be used to read and display one or more raw memory locations – in a cog register, in Hub RAM or in XMM RAM. If no specific location type is specified, it is assumed to be Hub RAM.

Note when writing or reading that the addresses shown on the Catalina listing are not necessarily the actual memory addresses. Catalina uses both a **relocation offset** and (for code addresses) a **CS** register that may affect the addresses used. The value of these can be determined using the **Mode** command.

The address to be read can also be specified by using **&variable** or **&function** – the read command will look up the address of the variable or function and read from that memory location – whether it is in Cog RAM, Hub RAM or XMM RAM, or at a specified offset from the current stack frame).

Note that an address can be specified using a variable name, or the address of a variable or function, but that you can use the resulting address to read from *either* Hub RAM *or* XMM RAM. This means it is possible to accidentally use a Hub RAM address to read XMM RAM or vice-versa It is therefore recommended to use the **Print** command instead of **Read** wherever possible.

Memory locations are always read as 4 byte (long) values.

The syntax is:

```
read location [ count ]
```

For example:

To read Cog RAM register SP:

```
read SP
```

or

```
r SP
```

To read 24 general purpose registers (R0 to R24) starting at R0:

```
read r0 24
```

or

```
r r0 24
```

To read 10 consecutive cog RAM locations starting at cog location 0x1:

```
read cog 0x1 10
```

or

```
r c 0x1 10
```

To read Hub RAM location 0x1000:

```
read hub 0x1000
```

or

```
r h 0x1000
```

or

```
r 0x1000
```

To read XMM RAM location 0x1000000

```
read xmm 0x1000000
```

or

```
r x 0x1000000
```

To read from Hub RAM using the *value* of a variable (which may be in XMM RAM or Hub RAM):

```
read xxx
```

or

```
read hub xxx
```

To read from XMM RAM using the *value* of a variable or function (which may be in XMM RAM or Hub RAM):

```
read xmm xxx
```

To read from XMM RAM using the *address* of a variable or function (which may be in XMM RAM or Hub RAM):

```
read xmm &xxx
```

To read from Hub RAM using the *address* of a variable or function (which may be in XMM RAM or Hub RAM):

```
read &xxx
```

or

```
read hub &xxx
```

To read 20 consecutive locations starting at the address of a variable or function:

```
read &xxx 20
```

or

```
r &xxx 20
```

Step External command

This command requests the debugger to '*step* to any line *external* to' the current function.

This command relies on virtual breakpoints – it deletes any virtual breakpoints (or user breakpoints) in the current function, then adds virtual breakpoints to every line *outside* the current function before continuing the program (which means that this command can take a long time to execute if used in a large program).

The main use for this command is when the **Step Up** function cannot be used, but it is desirable to return to the calling function – but the **Step Into** command was not used to get to the current line (which means the necessary virtual breakpoints have not been set in the calling function).

The downsides of using this command are that in a large program, it can take a long time to set up the necessary virtual breakpoints.

Note that **Step External** will stop at *any* line outside the current function – if the current function calls another function before returning, this command will stop in *that* function and not the calling function. For this reason, this command is best used when the current function is known to be about to return.

The syntax is:

```
step external
```

For example:

```
step ex
```

or

```
step e
```

or

```
s e
```

Step Up command

This command requests the debugger to 'step up' to the line following the line that resulted in the creation of the current stack frame - normally, this is the line following the line that called the current function. The debugger does not stop on any line in the current function, or any line called by the current function unless there are user breakpoints on those lines. If there are not, the debugger simply 'returns' from the current function.

This command relies on virtual breakpoints – it deletes any user breakpoints in the current function and all virtual breakpoints in any function. Then it adds a virtual breakpoint to every line in the function that called the current function before continuing the program (which means that this command can take some time to execute if used in a large program). The result is that the program will stop when the current function returns, unless there are user breakpoints set in other functions.

To identify the calling function, this command relies on the fact that every function call creates a new stack frame. However, this is only guaranteed to be true if the program was compiled using the **-g3** flag, because Catalina can 'optimize away' the need to create stack frames for small functions. This optimization is not disabled just because the **–g** flag has been specified. The result is that when debugging a program compiled using only **–g**, using 'Step Up' from a function that did not have a stack frame results in the debugger stepping up, and up again, until it finds a function that did construct a stack frame. To force a stack frame to be created, specify the option **–g3** instead - but note that this results in a larger program being generated.

If the **–g3** flag cannot be used (e.g. because the resulting program is too large or too slow), the **Step External** command provides an alternative to this command.

The syntax is:

```
step up
```

For example:

```
step up
```

or

```
step u
```

or

```
s u
```

Trace command

This command is used to print a stack frame trace that shows the sequence of function calls that resulted in the program arriving at the line at which it is currently stopped. The output is similar to that shown earlier (see the section **Line listing format**). The line at which the program is currently stopped is printed first, with the most recent line through which the program passed printed below that, then the older lines below that.

Note that this command relies on interpreting the stack frames, and therefore will only show *all* function calls if the program being debugged was compiled with the **-g3** option (which forces all functions to create a new stack frame). If the program is instead compiled only with **-g**, Catalina's normal optimizations may result in some functions not creating a stack frame. Calls from such functions will not be shown in the stack trace.

The syntax is:

```
trace
```

For example:

```
trace
```

or

```
t
```

Update command

The update command can be used to update the value of the named variable. The value is also printed.

The output of this command is affected by the setting of the verbose flag. See the Verbose command. If verbose is off, only the value is printed. If verbose is on, the data type and memory location of the variable are also printed.

The command only knows how to update scalar types (e.g. chars, ints, floats). Single array elements or structure components can be updated (by specifying the array index or component name), but not the whole array or structure. The type of value specified should match the type of the variable being updated. Array indexes can be variables names that are known in the program – BlackBox will fetch the current value of the variable when determining the address to update.

Note that only scalar variables can be updated, and only variables in scope of the line at which the debugger is currently stopped. To write arbitrary memory locations instead, see the **Write** command.

The syntax is:

```
update [ * ] variable_name [ = ] value
```

Multiple dereferencing operators (*) can be specified (e.g. if the variable holds "a pointer to a pointer to a pointer to …" etc).

For example, to update the value of the 3rd element of the array *dim* in the structure *coord* to 100:

```
update coord.dim[3] = 100
```

or

```
u coord.dim[3] 100
```

To update the location pointed to by this variable (assuming it is a pointer) to 100:

```
u *coord.dim[3] 100
```

To update the ***i***th element of the *dim* array to 100, where ***i*** is an integer variable in the program being debugged:

```
    u coord.dim[i] 100
```

To update program variables i, j, k, l, f with a specific type of value:

```
    u i short 0xffff
    u j byte 100
    u k char 'c'
    u l long 0xffffffff
    u f float 3.72
```

Verbose command

This command shows the current setting of the verbose flag, or turns it on or off. In verbose mode, more information is printed about variables, and about what the program is doing.

The syntax is:

```
    verbose [ off | on ]
```

For example:

To display the current setting of the flag:

```
    verbose
```

or

```
    v
```

To turn the flag on:

```
    verbose on
```

or

```
    v on
```

Write command

This command can be used to write a new value to a raw memory location – in a cog register, in Hub RAM or in XMM RAM. If no specific location type is specified, it is assumed to be Hub RAM.

Note that currently, memory locations can only be written with 4 byte types – but these do not need to be longs (e.g. they may be float values).

Note when writing or reading that the addresses shown on the Catalina listing are not necessarily the actual memory addresses. Catalina uses both a **relocation offset** and (for code addresses) a **CS** register that may affect the addresses used. The value of these can be determined using the **Mode** command.

Note that an address can be specified using a variable name, or the address of a variable or function, but that you can use the resulting address to write to *either* Hub RAM *or* XMM RAM. This means it is possible to accidentally use a Hub RAM address to write to XMM RAM or vice-versa It is therefore recommended to use the **Update** command instead of **Write** wherever possible.

The syntax of the command is:

```
write location [ = ] value
```

For example:

To write value 0xABCDFFFF Cog RAM register SP:

```
write SP = 0xABCDFFFF
```

or

```
w SP 0xABCDFFFF
```

To write value 1234 to Cog RAM location 0x1:

```
write cog 0x1 1234
```

or

```
w c 0x1 1234
```

To write -1 to Hub RAM location 0x1000:

```
write hub 0x1000 = -1
```

or

```
w h 0x1000 -1
```

or

```
w 0x1000 -1
```

To write 999 to Hub RAM using the *value* of a variable (which may be in XMM RAM or Hub RAM):

```
write xxx 999
```

or

```
write hub xxx 999
```

To write 999 to XMM RAM using the *value* of a variable or function (which may be in XMM RAM or Hub RAM):

```
write xmm xxx 999
```

To write 999 to XMM RAM using the *address* of a variable or function (which may be in XMM RAM or Hub RAM):

```
write xmm &xxx 999
```

To write 999 to Hub RAM using the *address* of a variable or function (which may be in XMM RAM or Hub RAM):

```
write &xxx 999
```

or

```
write hub &xxx 999
```

To write 1.234 XMM RAM location 0x1000000:

```
write xmm 0x1000000 float 1.234
```

or

```
w x 0x1000000 float 1.234
```

Xmm command

This command can be used to set the memory model. This may also be set on the command line during startup using the **-x** command line option.

NOTE: Normally it is normally not necessary to set the memory model – BlackBox will determine it automatically on startup by examining the program currently loaded into memory. Overriding this setting with a value different to the memory model specified when the program was compiled can cause the debug session to fail - this is because the debugger must use different methods to access the correct memory locations.

Memory models can be specified as **0** (or **tiny**), **2** (or **small**), **5** (or **large**), **8** (or **compact**), **100** (or **p2_tiny**), **102** (or **p2_small**), **105** (or **p2_large**) or **111** (or **p2_native**). Note that if the words rather than the numbers are used, they cannot be abbreviated.

If no parameters are specified, the current value of the memory model is displayed.

The syntax is:

```
xmm [ model ]
```

For example:

To set the memory model to **tiny**:

```
xmm tiny
```

or

```
x 0
```

To set the memory model to **p2_tiny**:

```
xmm p2_tiny
```

or

```
x 100
```

To display the memory model:

```
xmm
```

or

```
x
```

"." command

This command simply repeats the last command.

The syntax is:

```
.
```

Interrupt Command

This command can be used to regain control of the debugger if it is waiting for the program being debugged to reach the next breakpoint – e.g. if it is sitting in a loop waiting for some device or user input. Sometimes it is useful to be able to read and write hub RAM even under these circumstances. It is even possible to add breakpoints (applicable to LMM **tiny** mode programs only).

However, while the debugger is in this state many functions (such as reading variables, cogs or XMM RAM) will be unavailable – and it is not currently possible to interrupt the actual program being debugged, to force it back under the control of the debugger.

The syntax is:

```
Ctrl+C
```