

# Getting Started with the BlackBox Debugger

## Table of Contents

<a href="#">Introduction.....</a>	<a href="#">2</a>
<a href="#">What is BlackBox?.....</a>	<a href="#">2</a>
<a href="#">What is the status of BlackBox?.....</a>	<a href="#">2</a>
<a href="#">What are the prerequisites for BlackBox?.....</a>	<a href="#">2</a>
<a href="#">Why is BlackBox “unorthodox”?.....</a>	<a href="#">3</a>
<a href="#">Who should I contact about BlackBox?.....</a>	<a href="#">3</a>
<a href="#">Preparing programs for use with BlackBox.....</a>	<a href="#">4</a>
<a href="#">Overview.....</a>	<a href="#">4</a>
<a href="#">Preparing programs using the Catalina Command Line.....</a>	<a href="#">4</a>
<a href="#">Preparing programs using Catalina Geany.....</a>	<a href="#">6</a>
<a href="#">Loading programs from the Catalina Command Line.....</a>	<a href="#">6</a>
<a href="#">Loading programs from Catalina Geany.....</a>	<a href="#">6</a>
<a href="#">Starting BlackBox from the Catalina Command Line.....</a>	<a href="#">6</a>
<a href="#">Starting BlackBox from Catalina Geany.....</a>	<a href="#">7</a>
<a href="#">A Demonstration of BlackBox.....</a>	<a href="#">8</a>
<a href="#">The example program.....</a>	<a href="#">8</a>
<a href="#">Compiling and Loading the example.....</a>	<a href="#">8</a>
<a href="#">Compiling for Propeller 2 boards.....</a>	<a href="#">9</a>
<a href="#">Compiling for the C3.....</a>	<a href="#">9</a>
<a href="#">Compiling and Loading on the Hydra or Hybrid.....</a>	<a href="#">9</a>
<a href="#">Compiling and Loading on the DracBlade, Demo Board or TriBladeProp.....</a>	<a href="#">9</a>
<a href="#">Compiling for other Propeller 1 boards.....</a>	<a href="#">10</a>
<a href="#">Starting BlackBox to debug the example.....</a>	<a href="#">10</a>
<a href="#">Displaying source files.....</a>	<a href="#">11</a>
<a href="#">Single Stepping through the program.....</a>	<a href="#">13</a>
<a href="#">User Breakpoints.....</a>	<a href="#">15</a>
<a href="#">Printing and Updating Variables.....</a>	<a href="#">16</a>
<a href="#">Combining User and Virtual Breakpoints.....</a>	<a href="#">18</a>
<a href="#">What to do next?.....</a>	<a href="#">20</a>
<a href="#">Known Issues with BlackBox.....</a>	<a href="#">21</a>

## Introduction

This document provides a brief tutorial to get you up and running with the BlackBox debugger for the Catalina C compiler. It assumes a reasonable degree of familiarity with both the Propeller and with Catalina.

A full reference manual for **BlackBox Reference Manual** is also provided with Catalina, containing more detail about various BlackBox commands and features. However, it is not necessary to read the reference manual before starting this tutorial.

For the sake of brevity, this document assumes you are using both Catalina and BlackBox under Windows. If you are using Linux, the commands and command options are identical, but some file and directory names may have to be changed

### ***What is BlackBox?***

BlackBox is a command line application that allows source level debugging of C programs that have been compiled using the Catalina C compiler, and which are loaded and executed on a supported Propeller platform.

BlackBox runs on the PC, and communicates via a serial connection with the program being debugged.

BlackBox provides the ability to display C source files, set and clear user breakpoints, single step through C programs (BlackBox uses a slightly unorthodox method of doing this), and display or modify the values of any local or global C variables.

### ***What is the status of BlackBox?***

BlackBox is capable of debugging arbitrary C programs, including all the demonstration programs provided with Catalina.

BlackBox supports Propeller 1 and Propeller 2 programs.

BlackBox supports debugging of CMM, LMM and XMM programs on all Propeller 1 platforms supported by Catalina (with the exception of XMM programs executed from SPI Flash – such programs must be recompiled to use SPI RAM only).

BlackBox supports debugging of CMM, LMM, XMM & NMM programs on all Propeller 2 platforms supported by Catalina.

### ***What are the prerequisites for BlackBox?***

BlackBox has been tested under Windows XP SP2 (32 bit) and Windows 7 (64 bit), but should also run on other versions of Windows.

BlackBox has been tested under Linux Fedora, Debian and Ubuntu distribution. It should also run under other Linux distributions.

Programs to be debugged using the current release of BlackBox must be compiled using Catalina release 4.0 or greater – the current version of BlackBox may not work with any earlier versions of Catalina (instead, use the version that came with the version of Catalina you are using). Programs must be prepared by compiling them using a new Catalina **-g** or **-g3** command line options (the meaning and use of these options is described later in this document).

BlackBox requires that the program being debugged has a cog free to implement the BlackBox serial protocol. Other than that, the overhead of BlackBox on the compiled

program is limited to the execution of approximately 100 additional instructions (400 bytes) during program initialization. When not actually stopped at a breakpoint, the program executes as usual (i.e. at normal speed) and is unaware of BlackBox – this makes BlackBox suitable for debugging real-time or time-critical programs.

BlackBox also requires a USB or RS232 serial connection to the Propeller running the program to be debugged. On Propeller 1 platforms that use the HX512 SRAM expansion card (e.g. the Hydra and the Hybrid), a special cable is required – this is the same cable as is used for the Catalina Payload program loader – see the **Catalina Reference Manual** for more detail. On other Propeller 1 platforms or Propeller 2 platforms a PropPlug can be used.

### ***Why is BlackBox “unorthodox”?***

The BlackBox debugger uses a slightly unusual approach to breakpoints, which can be of two types – user or virtual. This can sometimes lead to unexpected results when a program is executed using a combination of manual breakpoints, and the various single step operations (step next, step into, step out) which use virtual breakpoints.

This is more fully explained later in this document.

### ***Who should I contact about BlackBox?***

Contact Ross Higson at [ross@thevastydeep.com](mailto:ross@thevastydeep.com)

## Preparing programs for use with BlackBox

### Overview

Preparing a C program can be as simple as including the **-g** or **-g3** command line options when compiling the program with Catalina. This instructs Catalina to do two things:

1. Generate additional information that is required by the BlackBox debugger. A **.debug** file is generated for each C source file, and these are then combined into one **.dbg** file that can be loaded by BlackBox (the **.debug** files can then be deleted).
2. Select a BlackBox enabled target for the program. Suitable targets are provided for CMM, LMM and XMM programs. These targets contain additional initialization code that loads the debug cog, and performs other additional tasks required to prepare the kernel for co-operation with the debug cog.

In addition, the **-g3** option does the following:

3. Disable some compiler optimizations that can make debugging more complex. For example, this option ensures that all functions have a stack frame whether they need it or not, which simplifies single stepping programs. While using this option can increase program size and execution time slightly, it is recommended, unless it makes the program too large to load, or execute too slowly.

There are a few things that you need to be aware of when preparing programs for use with BlackBox:

The program must have a spare cog available (to run the debug cog).

The program must have two pins (normally Propeller pins 30 and 31, but others can be used) available for serial communication with the BlackBox debugger running on the host.

You cannot specify your own target – to use BlackBox you must let Catalina select a BlackBox-enabled target.

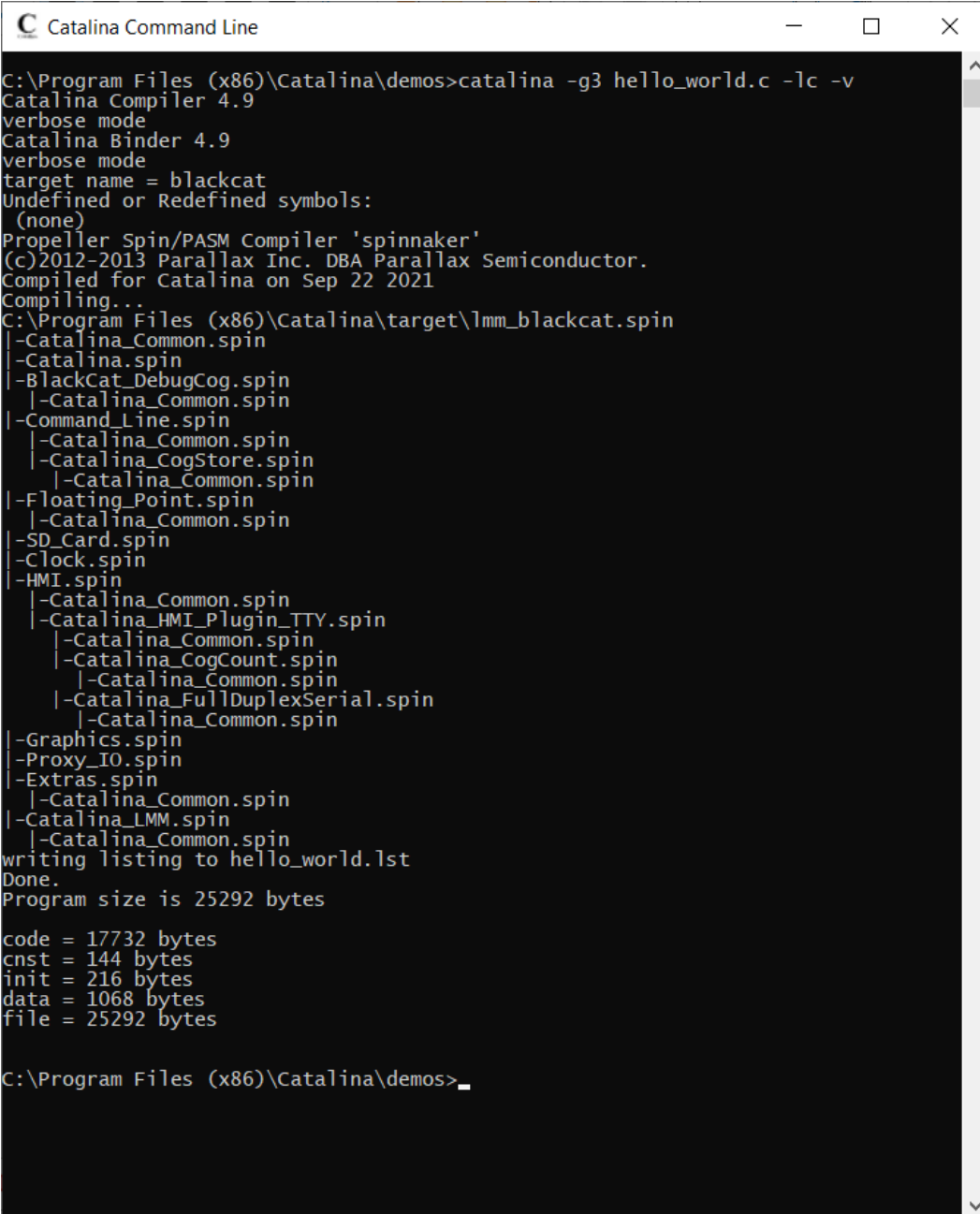
The resulting program executable will be slightly larger – about 400 additional bytes of initialization code, plus up to 512 bytes of debug cog code (however, the space used for the debug cog code is reclaimed during initialization for use as program stack or heap space once the program is executing).

### Preparing programs using the Catalina Command Line

As mentioned above, simply include the **-g** or **-g3** command line options along with all the other normal options when compiling the program. For example, after setting up to use Catalina (using the **use\_catalina** batch file) execute the following command in the Catalina demo directory:

```
catalina -g3 hello_world.c -lc -v
```

The **-v** flag will cause Catalina to produce verbose output. This command will produce output similar to the following:

A screenshot of a Windows command prompt window titled "Catalina Command Line". The window has a black background with white text. The command entered is "catalina -g3 hello\_world.c -lc -v". The output shows the Catalina Compiler 4.9 in verbose mode, using the Imm\_blackcat.spin target. It lists various source files being compiled, including Catalina\_Common.spin, Catalina.spin, BlackCat\_DebugCog.spin, and many others. It also shows the program size (25292 bytes) and a breakdown of code, constants, init, data, and file sizes. The prompt ends with "C:\Program Files (x86)\Catalina\demos>\_".

```
Catalina Command Line
C:\Program Files (x86)\Catalina\demos>catalina -g3 hello_world.c -lc -v
Catalina Compiler 4.9
verbose mode
Catalina Binder 4.9
verbose mode
target name = blackcat
Undefined or Redefined symbols:
(none)
Propeller Spin/PASM Compiler 'spinnaker'
(c)2012-2013 Parallax Inc. DBA Parallax Semiconductor.
Compiled for Catalina on Sep 22 2021
Compiling...
C:\Program Files (x86)\Catalina\target\Imm_blackcat.spin
|-Catalina_Common.spin
|-Catalina.spin
|-BlackCat_DebugCog.spin
|-Catalina_Common.spin
|-Command_Line.spin
|-Catalina_Common.spin
|-Catalina_CogStore.spin
|-Catalina_Common.spin
|-Floating_Point.spin
|-Catalina_Common.spin
|-SD_Card.spin
|-Clock.spin
|-HMI.spin
|-Catalina_Common.spin
|-Catalina_HMI_Plugin_TTY.spin
|-Catalina_Common.spin
|-Catalina_CogCount.spin
|-Catalina_Common.spin
|-Catalina_FullDuplexSerial.spin
|-Catalina_Common.spin
|-Graphics.spin
|-Proxy_IO.spin
|-Extras.spin
|-Catalina_Common.spin
|-Catalina_LMM.spin
|-Catalina_Common.spin
writing listing to hello_world.lst
Done.
Program size is 25292 bytes

code = 17732 bytes
cnst = 144 bytes
init = 216 bytes
data = 1068 bytes
file = 25292 bytes

C:\Program Files (x86)\Catalina\demos>_
```

We added the **-v** option here just to show that for this compilation, Catalina has automatically used the **Imm\_blackcat.spin** target instead of the more usual **Imm\_default.spin** target (for historical reasons BlackBox targets are called BlackCat).

In addition to a **hello\_world** executable, Catalina will also generate the following files:

- A file called **hello\_world.lst** (a listing file is *a*lways produced when the **-g** or **-g3** command-line options are used).

- A file called **hello\_world.debug** (one such file is generated file for each C source file, named with the same name as the C source file but with a **.debug** extension)

- A file called **hello\_world.dbg** (one such file is generated for the whole

compilation, named with the same name as the output binary file but with a **.dbg** extension).

Note that the files actually required for debugging are the **hello\_world.binary** (or **hello\_world.bin** on the Propeller 2) and the **hello\_world.dbg** file. The other files are left for reference only, and can be deleted once the compilation is complete. However, also note that it is *essential* that the correct dbg file is used when debugging – this file contains information about the location in the binary file of various program elements. If the program is modified and recompiled, the .dbg file *must* be regenerated by including the **-g** or **-g3** option.

### ***Preparing programs using Catalina Geany***

You can easily compile and debug programs from within Geany. Just add the **-g3** flag to the project build options. Then simply compile and link your program as normal.

### ***Loading programs from the Catalina Command Line***

There are various methods for loading programs into the Propeller, including the following:

1. Using the Catalina Payload program loader. This program supports CMM, LMM and XMM programs on the Propeller 1, and CMM, LMM and NMM programs on the Propeller 2. Refer to the **Catalina Reference Manual** for more details.
2. Using the Parallax Propeller tool (CMM and LMM programs on the Propeller 1 only).
3. For EMM programs, program them into the EEPROM of the Propeller.
4. For SMM programs, write them onto an SD Card and load them via Catalyst.
5. For CMM, LMM or XMM programs, write them to an SD card and then use the Catalina Generic SD Program Loader (for multi-CPU systems such as the TriBladeProp, you may also need to use the Catalina Generic SIO Program Loader)

For example, to automatically locate the first port with a Propeller attached, and then load an executable called **program**, the following payload command might be used:

```
payload program
```

In any case, all programs built using the **-g** or **-g3** options always stop after the execution of the initialization code, and wait till the BlackBox debugger establishes serial communications with the debug cog.

### ***Loading programs from Catalina Geany***

From the **Build** menu, select:

**Download and Execute**

### ***Starting BlackBox from the Catalina Command Line***

Starting BlackBox to debug an executable program like can be done by entering a command similar to the following:

```
blackbox program
```

This command accepts parameters described in the **BlackBox Reference Manual**.

BlackBox should automatically determine the port to use for debugging – even on the Hydra or Hybrid platforms where a special debug cable is required. If it does not, the port to use can be specified via the **-p** option. For example:

```
blackbox -p2 program
```

Note that in BlackBox, the **-p** option specifies the port. It is not necessary to specify whether a Propeller 1 or a Propeller 2 is in use – BlackBox detects this automatically.

**Note** that under Linux, starting BlackBox may automatically raise DTR, which can cause the Propeller to reset. If this occurs on your platform, it is necessary to temporarily disable the RESET connection – see the **Known Issues** section for more details.

### ***Starting BlackBox from Catalina Geany***

From the **Build** menu, select:

#### **Debug with BlackBox**

This will start the BlackBox debugger in a command window.

## A Demonstration of BlackBox

This section provides a real-world demonstration of BlackBox. It uses an example program provided with the Catalina distribution, in the **demos\debug** subdirectory.

### *The example program*

The example program requires an external VGA or TV display and a keyboard, but no mouse or floating point libraries.

The example program consists of the following C source files:

```
debug_example.h  
debug_example.c  
debug_functions_1.c  
debug_functions_2.c
```

The example program performs various test functions designed to explore the capabilities of BlackBox. Most of the functions print their results on the display, and some require user interaction via the keyboard.

### *Compiling and Loading the example*

A batch file is provided for building the program, and a **Geany** project is also provided, which by default builds for the **C3** – if you plan to use **Geany** but you do not have a C3 then you will need to edit the project options to suit your platform.

To compile the example program using the command line, open a Catalina Command Line window. Then go to the *demos\debug* directory (or copy that directory to our own home directory).

There is a **build\_all** script in this folder to compile the program. See the sections below for notes on compiling for various specific platforms.

Compiling the program will generate various files similar to the following:

```
debug_example.binary  
debug_example.dbg  
debug_example.lst  
debug_example.debug  
debug_functions1.debug  
debug_functions2.debug
```

In this demonstration, we will use the binary file and the dbg file. The others are generated for information purposes whenever the **-g** or **-g3** option is specified (in this case the **-g3** option is included by the scripts that we will use to build the example program).

Then load the resulting **debug\_example** executable file into the Propeller using the Catalina Payload program loader:

```
payload debug_example
```

or

```
payload -pN debug_example
```

Where **N** is the normal serial port connected to the Propeller for loading programs.

On the Propeller 1, BlackBox expects to use the serial port on pins 30 & 31. On the Propeller 2, BlackBox expects to use the serial port on pins 62 & 63. If you want to use other pins (e.g. because your program uses a serial HMI on those pins) then you will



have to modify the pins used by BlackBox and also use a Prop Plug on those pins – see the **BlackBox Reference Manual** for more details.

When the binary has been loaded, all the drivers are initialized, so you might see a blank display appear because the plugins will be started as normal. However, the C program will not begin to execute until we start BlackBox.

## Compiling for Propeller 2 boards

All Propeller 2 boards default to using a serial HMI option, so you can build the program using the command **build\_all** in the *demo\debug* directory, but you need to either specify a non-serial HMI option (e.g. VGA) or else modify the platform to tell BlackBox to use other pins.

For example:

```
build_all P2_EVAL VGA
```

## Compiling for the C3

If you have a C3, then you can build the program using the command **build\_all** in the *demo\debug* directory, specifying the C3 as a command line parameter. You can also specify the HMI option to use (e.g. VGA) if you do not want the default (which is to use the TV output).

For example:

```
build_all C3
```

or

```
build_all C3 HIRES_VGA
```

## Compiling and Loading on the Hydra or Hybrid

If you have a Hydra and Hybrid (with or without the HX512 SRAM card) you will need a special cable plugged into the Propeller mouse port, and you must also build the demo program to exclude the normal mouse driver. For example, use the **build\_all** command as follows:

```
build_all HYBRID NO_MOUSE
```

or:

```
build_all HYDRA NO_MOUSE
```

The special cable you will need is described in the **Catalina Reference Manual**. A normal serial cable is used to load the program, but the default debugging target for the Hybrid and Hydra assumes that the mouse port will be used for debugging programs.

## Compiling and Loading on the DracBlade, Demo Board or TriBladeProp

If you have a DracBlade, Demo board or TriBladeProp, then you can build the program using the command **build\_all** in the *demo\debug* directory, specifying the Propeller platform as a command line parameter.

If you have a TriBladeProp, you must also specify the CPU – the demo program requires both a screen and keyboard device, so is most easily built for CPU #1. For example:

```
build_all DRACBLADE
```

or

```
build_all DEMO
```

or

```
build_all TRIBLADEPROP CPU_1
```

When loading the TriBladeProp, make sure the jumpers and cables are set to connect to CPU #1.

## Compiling for other Propeller 1 boards

If you have another board, use the **build\_all** in the demo\debug directory, specifying your board (or CUSTOM) as a command line parameter. You can also specify the HMI option to use (e.g. VGA) if your board defaults to a serial HMI.

For example:

```
build_all CUSTOM
```

or

```
build_all CUSTOM HIRES_VGA
```

## Starting BlackBox to debug the example

Start BlackBox using the command:

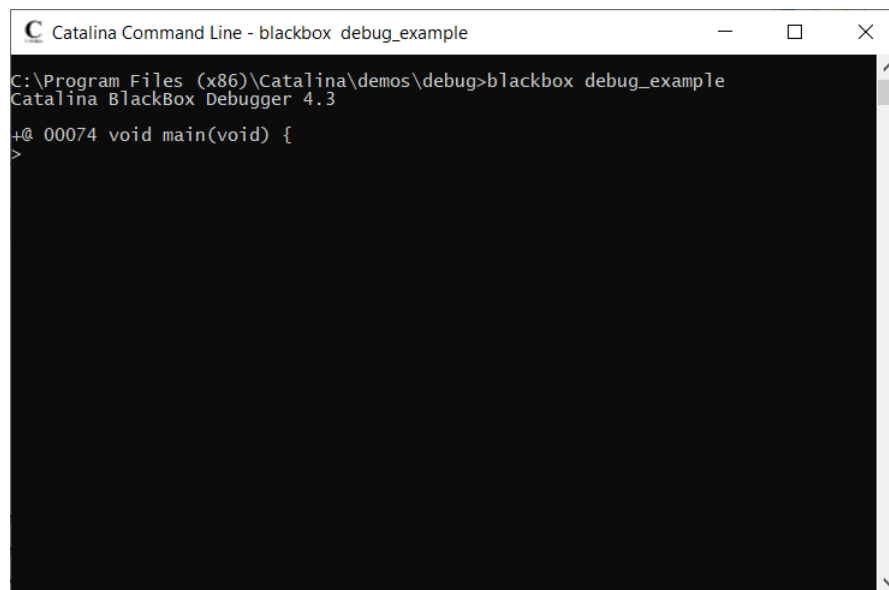
```
blackbox debug_example
```

or

```
blackbox -pN debug_example
```

Where **N** is the port connected to the Propeller mouse port (Hybrid or Hydra only), or to the debug port configured for your platform (which by default is pins 30 and 31 for Propeller 1 platforms, or pins 62 & 63 for Propeller 2 platforms).

For example, here is how your command window might look after using payload, and starting blackbox:



```
Catalina Command Line - blackbox debug_example
C:\Program Files (x86)\Catalina\demos\debug>blackbox debug_example
Catalina BlackBox Debugger 4.3
+@ 00074 void main(void) {
>
```

BlackBox is now waiting for commands, and indicating that the program is stopped at line 74 – which is the first line of the function main.

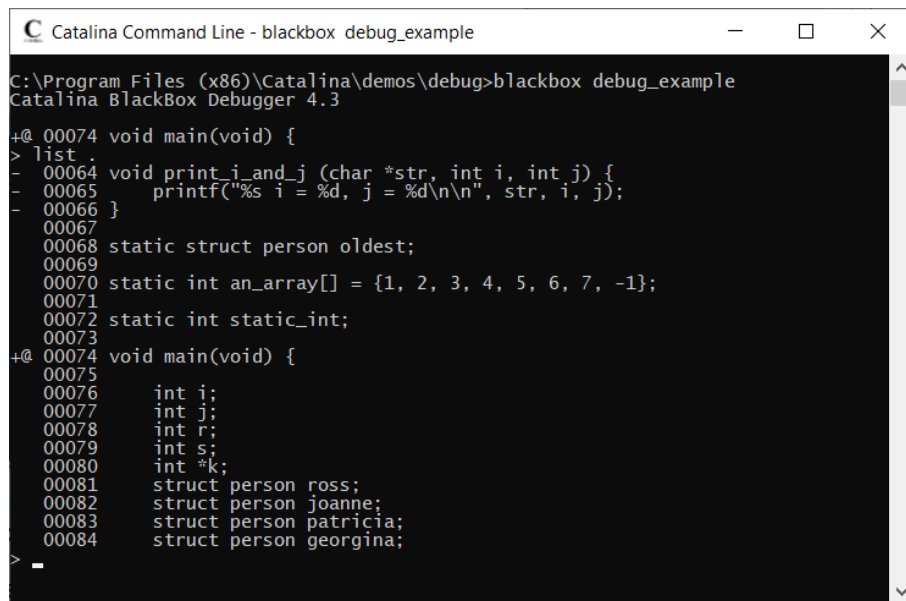
The + at the start of the line indicates that a virtual breakpoint has been placed on this line, and the @ indicates this is the source line at which the debugger is currently stopped.

### ***Displaying source files***

The first thing we will do is display the source around the point we are currently stopped. To do this, we use the **List** command, specifying the parameter '.' which refers to the current debug location. Enter the following command (note there is a space between the 'list' and the '.')

```
list .
```

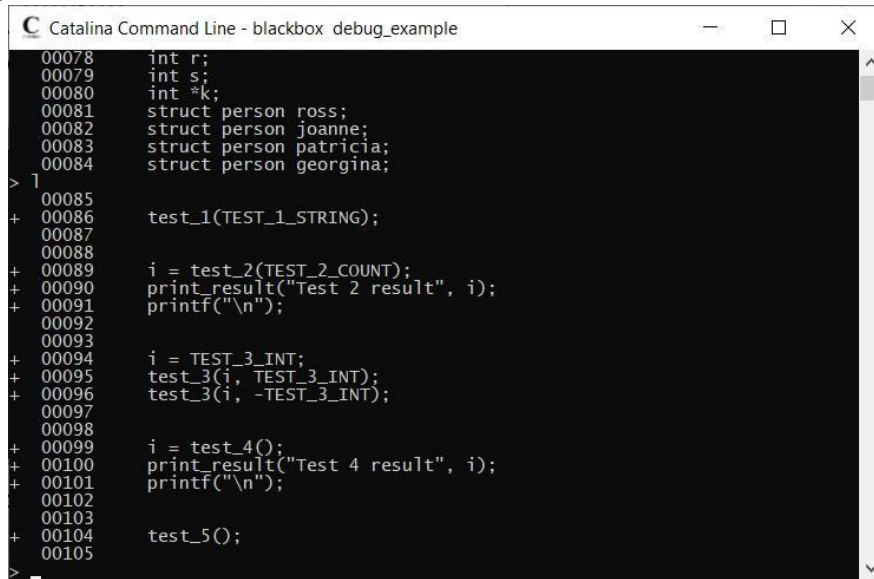
The result should look similar to the following:



```
Catalina Command Line - blackbox debug_example
C:\Program Files (x86)\Catalina\demos\debug>blackbox debug_example
Catalina BlackBox Debugger 4.3
+@ 00074 void main(void) {
> list .
- 00064 void print_i_and_j (char *str, int i, int j) {
- 00065     printf("%s i = %d, j = %d\n\n", str, i, j);
- 00066 }
00067
00068 static struct person oldest;
00069
00070 static int an_array[] = {1, 2, 3, 4, 5, 6, 7, -1};
00071
00072 static int static_int;
00073
+@ 00074 void main(void) {
00075
00076     int i;
00077     int j;
00078     int r;
00079     int s;
00080     int *k;
00081     struct person ross;
00082     struct person joanne;
00083     struct person patricia;
00084     struct person georgina;
```

The source lines 64 to 84 are displayed (i.e. 20 lines centered on the current debug location). The – at the beginning of lines 64 to 66 indicate that there is code associated with these lines. Other than line 62, all the other lines are type and variable declarations, and have no actual code associated with them. This is significant because only lines with code can have a breakpoint set on them.

To continue looking at the source, use the 'list' command again, but this time with no parameter (and we can abbreviate it to just 'l'). The result should now look similar to the following:



```

Catalina Command Line - blackbox debug_example
00078     int r;
00079     int s;
00080     int *k;
00081     struct person ross;
00082     struct person joanne;
00083     struct person patricia;
00084     struct person georgina;
> l
00085
00086     test_1(TEST_1_STRING);
00087
00088
00089     i = test_2(TEST_2_COUNT);
00090     print_result("Test 2 result", i);
00091     printf("\n");
00092
00093
00094     i = TEST_3_INT;
00095     test_3(i, TEST_3_INT);
00096     test_3(i, -TEST_3_INT);
00097
00098
00099     i = test_4();
00100     print_result("Test 4 result", i);
00101     printf("\n");
00102
00103
00104     test_5();
00105

```

You can continue listing this source, or list a specific line or function. For example, try any of the following:

```

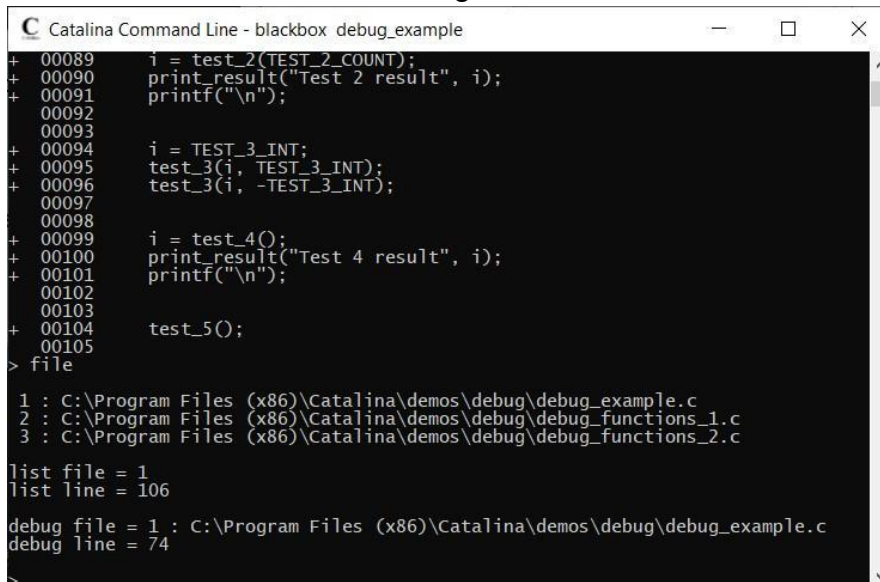
list test_1
list 50
list 20-30

```

You can also list sources in other files. To see the list of source files that comprise the current program, use the **File** command. Enter the command:

```
file
```

The result should look similar to the following:



```

Catalina Command Line - blackbox debug_example
+ 00089     i = test_2(TEST_2_COUNT);
+ 00090     print_result("Test 2 result", i);
+ 00091     printf("\n");
+ 00092
+ 00093
+ 00094     i = TEST_3_INT;
+ 00095     test_3(i, TEST_3_INT);
+ 00096     test_3(i, -TEST_3_INT);
+ 00097
+ 00098
+ 00099     i = test_4();
+ 00100     print_result("Test 4 result", i);
+ 00101     printf("\n");
+ 00102
+ 00103
+ 00104     test_5();
+ 00105
> file
1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
2 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_1.c
3 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_2.c
list file = 1
list line = 106
debug file = 1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
debug line = 74
>

```

This shows the three C source files that comprise the current program (the debugger does not know about .h files, only .c files), and also indicates the current list line and the current debug line. The File command can also be used to change files, so that you can list lines in a different file.

For example, try:

```
file 3
list 50
```

No matter which file you are currently listing, the command ‘file .’ will always set the current list file and line to be the current debug location, and the command ‘list .’ will always list the current debug location.

In the remainder of this tutorial, we will always show all commands entered in full – but all commands can be abbreviated, usually down to a single character. For example, the following are all equivalent:

```
list 50
lis 50
li 50
l 50
```

## ***Single Stepping through the program***

Now we’ll try executing a line or two of the program. There are two different commands that can be used to execute the current line of code:

**Step Next**      execute the current line of code and step to the next line in the current function that contains executable code. Can also be abbreviated to either **Step** or **Next**.

**Step Into**      execute the current line of code, but if there is a function call in this line, step *into* that function and stop at the function entry point. Otherwise, this command also just steps to the next line in the current function. Can also be abbreviated to **Into**.

When you are single stepping *within* a function, there are various ways to get *out* again:

**Step Out**      execute lines until the program calls another function, or reaches any existing breakpoint. The effect of this is that if the current function was reached via ‘Step Into’ then this will return to the calling function. Can also be abbreviated to just **Out**.

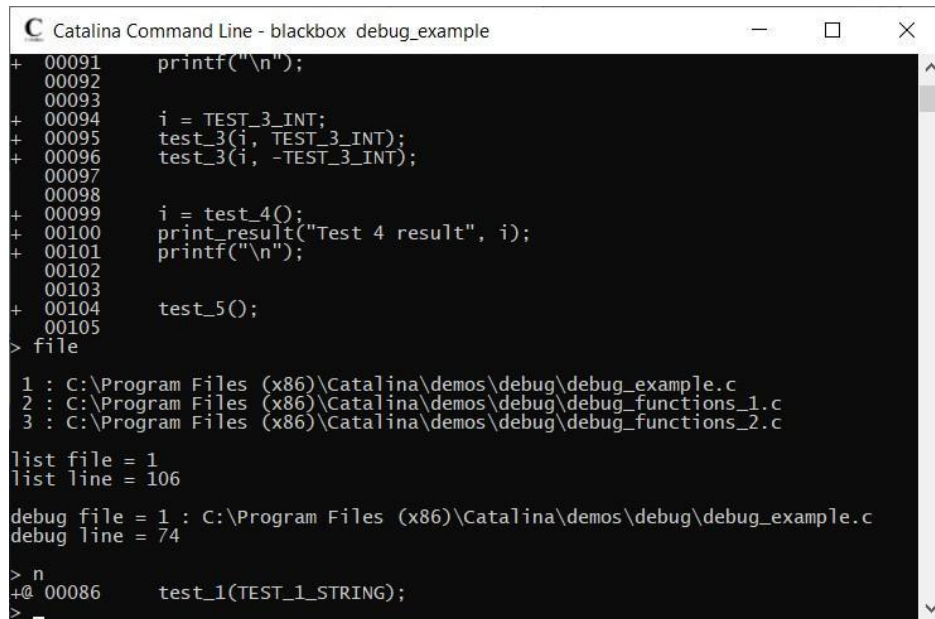
**Step External**   execute lines until the program reaches *any* line of code *external to the current function*. This may be the next line in the function that called the current function, or a line in a function that the current function calls.

**Step Up**      execute lines until the program returns from the current function. Actually, this returns to the call prior to the function call where the current stack frame was created. This will always be the calling function if the **-g3** command line option was used. If the **-g** command line option was used instead, this may cause the program to return to a function call further up the calling hierarchy, not the immediately preceding function call.

To single step one line, enter the **Next** command:

```
next
```

The result should look similar to the following, indicating we have stepped to the next line containing executable code, which is line 86:



```

Catalina Command Line - blackbox debug_example
+ 00091    printf("\n");
00092
00093
+ 00094    i = TEST_3_INT;
+ 00095    test_3(i, TEST_3_INT);
+ 00096    test_3(i, -TEST_3_INT);
00097
00098
+ 00099    i = test_4();
+ 00100    print_result("Test 4 result", i);
+ 00101    printf("\n");
00102
00103
+ 00104    test_5();
00105
> file
1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
2 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_1.c
3 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_2.c

list file = 1
list line = 106

debug file = 1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
debug line = 74

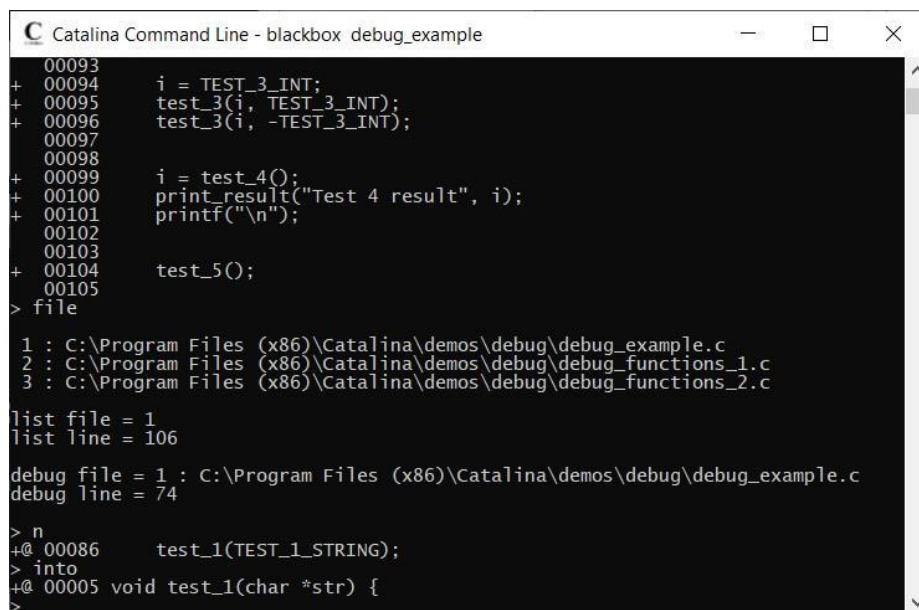
> n
+@ 00086    test_1(TEST_1_STRING);
>

```

This line contains a function call, so we enter the Step Into command:

**into**

The result will show we have stepped into the function test\_1:



```

Catalina Command Line - blackbox debug_example
+ 00093    i = TEST_3_INT;
+ 00094    test_3(i, TEST_3_INT);
+ 00095    test_3(i, -TEST_3_INT);
00096
00097
+ 00098    i = test_4();
+ 00099    print_result("Test 4 result", i);
+ 00100    printf("\n");
00101
00102
+ 00103    test_5();
00104
> file
1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
2 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_1.c
3 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_2.c

list file = 1
list line = 106

debug file = 1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
debug line = 74

> n
+@ 00086    test_1(TEST_1_STRING);
> into
+@ 00005    void test_1(char *str) {
>

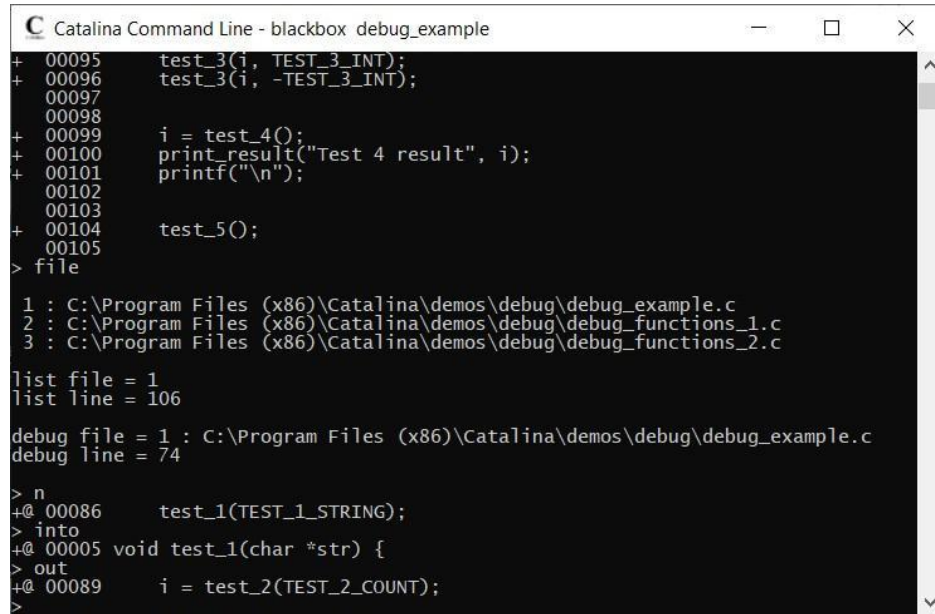
```

At this point we have not actually started executing the function – we are stopped at the entry point to it. However, this function doesn't do anything interesting, so now that we know how to get into a function, let's just get out again. There are several commands we could use, but in this case the **Out** command is the simplest. So enter the command:

**out**



The result will show we have stepped back to line 77 of the **main** function:



```

C Catalina Command Line - blackbox debug_example
+ 00095 test_3(i, TEST_3_INT);
+ 00096 test_3(i, -TEST_3_INT);
00097
00098
+ 00099 i = test_4();
+ 00100 print_result("Test 4 result", i);
+ 00101 printf("\n");
00102
00103
+ 00104 test_5();
00105
> file

1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
2 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_1.c
3 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_2.c

list file = 1
list line = 106

debug file = 1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
debug line = 74

> n
+@ 00086 test_1(TEST_1_STRING);
> into
+@ 00005 void test_1(char *str) {
> out
+@ 00089 i = test_2(TEST_2_COUNT);
>

```

The result of the `test_1` function we just exited should also appear on the display connected to the Propeller.

We can single step through more functions, but it is often useful when debugging to be able to go straight to a particular line of code. To do this we use user breakpoints. This is the subject of the next section.

## User Breakpoints

Sometimes, we just want to set a user breakpoint on a specific line and let the program execute until it reaches it.

There are several commands for dealing with user breakpoints:

**Breakpoint** add a user breakpoint to a specific line. A function name can also be specified to add a breakpoint to the first line of a function. If no parameters are specified this command lists all current user breakpoints.

**Delete** remove a user breakpoint from a specified line.

**Continue** Let the program execute until it reaches a user breakpoint.

**Go** Same as Continue.

In this case, we will set a breakpoint on line 147, and then let the program continue execution until it reaches this user breakpoint.

To do that, enter the commands:

```

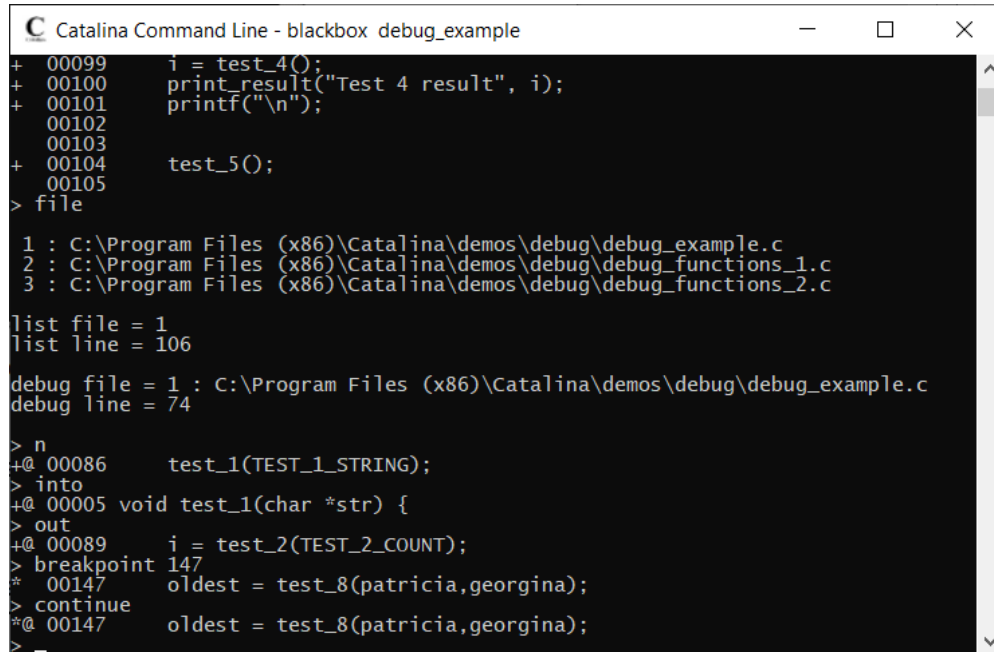
breakpoint 147
continue

```

Note that the example program we are using requires keyboard input before it will reach line 147 – after entering the **continue** command, at least two keys must be pressed on the keyboard connected to the Propeller to make the program stop again (any two keys will do).

Note also that there may be a pause after entering the **continue** command while the program removes the virtual breakpoints it has previously placed (to support single stepping) – this is characteristic of BlackBox, and can make it appear to take several seconds to respond to some commands.

When the program has stopped at line 147, the result will appear as follows:



```

Catalina Command Line - blackbox debug_example
+ 00099      i = test_4();
+ 00100      print_result("Test 4 result", i);
+ 00101      printf("\n");
+ 00102
+ 00103
+ 00104      test_5();
+ 00105
> file

1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
2 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_1.c
3 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_2.c

list file = 1
list line = 106

debug file = 1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
debug line = 74

> n
+@ 00086      test_1(TEST_1_STRING);
> into
+@ 00005 void test_1(char *str) {
> out
+@ 00089      i = test_2(TEST_2_COUNT);
> breakpoint 147
* 00147      oldest = test_8(patricia,georgina);
> continue
+@ 00147      oldest = test_8(patricia,georgina);
>

```

In the next section, we will display and modify some program variables.

## Printing and Updating Variables

A powerful feature of BlackBox is its ability to display and modify local variables. To do this we use the Print and Update commands:

**Print** This command accepts the name of a variable, and prints its value. The address of the variable can also be printed by prefixing it with **&**, and (if the variable is a pointer) the memory it points to can also be printed by prefixing it with **\***. If the variable is a structure, each field of the structure will be printed. If the variable is an array, an array index can be included – which can be an integer variable.

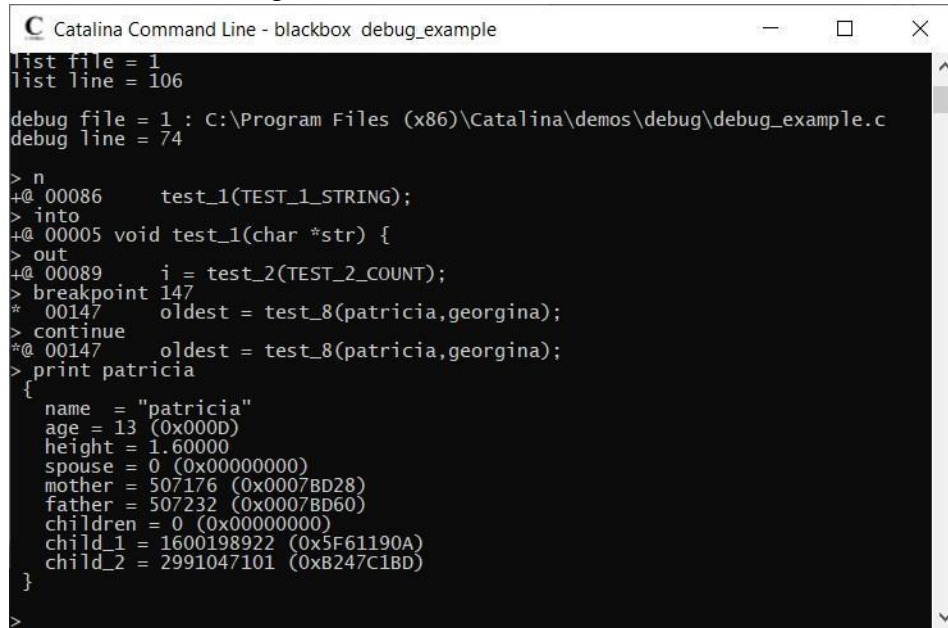
**Update** This command can be used to update the value of a simple scalar variable (or a field of a structure, or an element of an array).

Line 147 calls the function **test\_8**, which compares the value of two local variables – called **patricia** and **georgina**. To display the value of **patricia**, enter the command:

```
print patricia
```



The result will look something like:



```

Catalina Command Line - blackbox debug_example
list file = 1
list line = 106

debug file = 1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
debug line = 74

> n
+@ 00086      test_1(TEST_1_STRING);
> into
+@ 00005 void test_1(char *str) {
> out
+@ 00089      i = test_2(TEST_2_COUNT);
> breakpoint 147
* 00147      oldest = test_8(patricia,georgina);
> continue
+@ 00147      oldest = test_8(patricia,georgina);
> print patricia
{
  name = "patricia"
  age = 13 (0x000D)
  height = 1.60000
  spouse = 0 (0x00000000)
  mother = 507176 (0x0007BD28)
  father = 507232 (0x0007BD60)
  children = 0 (0x00000000)
  child_1 = 1600198922 (0x5F61190A)
  child_2 = 2991047101 (0xB247C1BD)
}

```

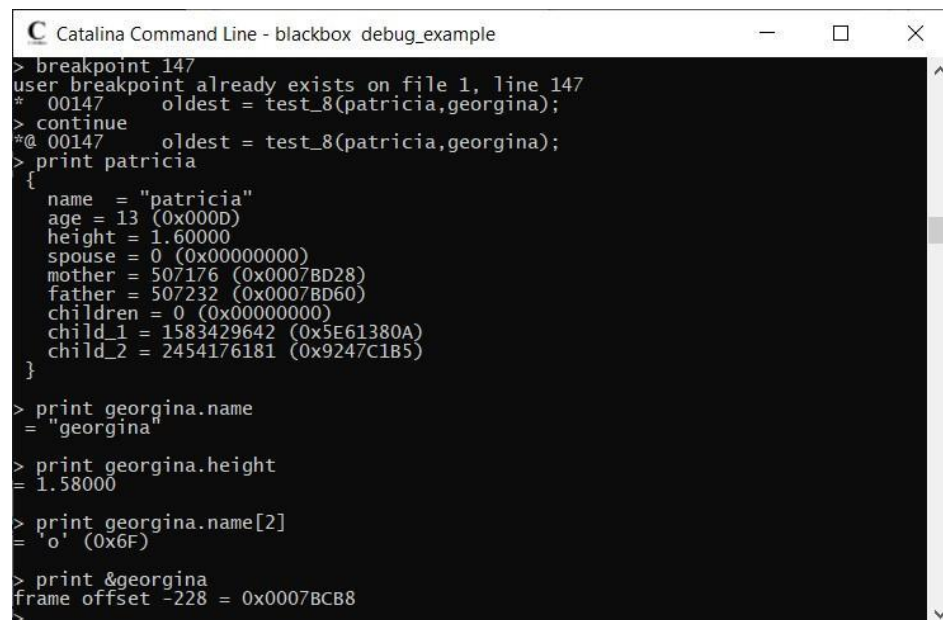
In this case, the variable is a structure, so all the fields are printed. To print individual fields, try the following commands:

```

print georgina.name
print georgina.height
print georgina.name[2]
print &georgina

```

The results will look something like:



```

Catalina Command Line - blackbox debug_example
> breakpoint 147
user breakpoint already exists on file 1, line 147
* 00147      oldest = test_8(patricia,georgina);
> continue
+@ 00147      oldest = test_8(patricia,georgina);
> print patricia
{
  name = "patricia"
  age = 13 (0x000D)
  height = 1.60000
  spouse = 0 (0x00000000)
  mother = 507176 (0x0007BD28)
  father = 507232 (0x0007BD60)
  children = 0 (0x00000000)
  child_1 = 1583429642 (0x5E61380A)
  child_2 = 2454176181 (0x9247C1B5)
}

> print georgina.name
= "georgina"

> print georgina.height
= 1.58000

> print georgina.name[2]
= 'o' (0x6F)

> print &georgina
frame offset -228 = 0x0007BCB8

```

The last one is a bit unusual – in this case, BlackBox is telling us that the address of **georgina** is in the current stack frame, at offset -228. Variables may be in registers, in the stack frame or at an absolute location in memory. It makes no difference when printing or updating them – BlackBox knows how to do all these things.

To update the values, enter the following commands:

```

update patricia.height = 1.9

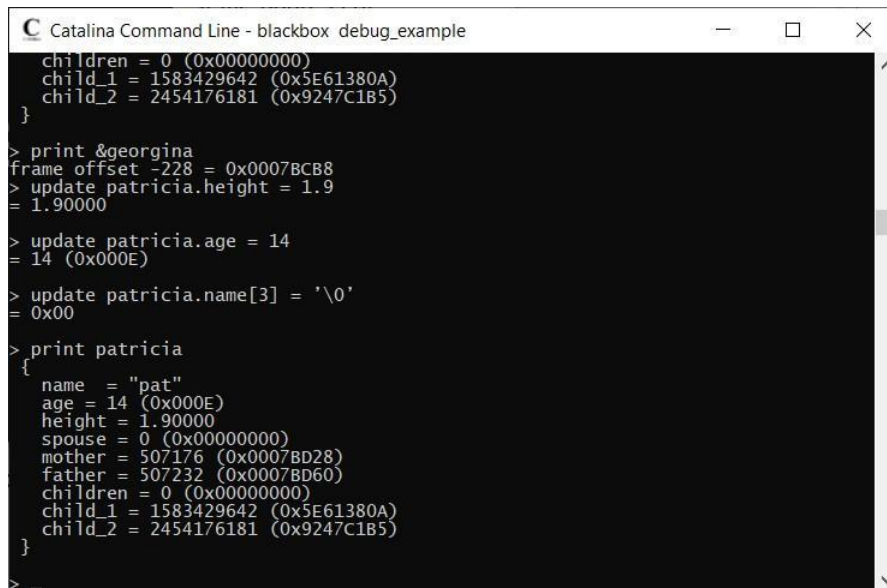
```

```

update patricia.age = 14
update patricia.name[3] = '\0'
print patricia

```

The results will look something like:



```

Catalina Command Line - blackbox debug_example
children = 0 (0x00000000)
child_1 = 1583429642 (0x5E61380A)
child_2 = 2454176181 (0x9247C1B5)
}
> print &georgina
frame offset -228 = 0x0007BCB8
> update patricia.height = 1.9
= 1.90000
> update patricia.age = 14
= 14 (0x000E)
> update patricia.name[3] = '\0'
= 0x00
> print patricia
{
  name = "pat"
  age = 14 (0x000E)
  height = 1.90000
  spouse = 0 (0x00000000)
  mother = 507176 (0x0007BD28)
  father = 507232 (0x0007BD60)
  children = 0 (0x00000000)
  child_1 = 1583429642 (0x5E61380A)
  child_2 = 2454176181 (0x9247C1B5)
}
>

```

Note that only variables that would normally be 'visible' to the program from its current location can be printed or updated.

The next section points out some of the complexities when combining user breakpoints with the virtual breakpoints used to support the single step functions.

## Combining User and Virtual Breakpoints

While working through this example, we have used both user breakpoints (which we set and cleared on specific lines) and virtual breakpoints (via the single step operations). However, care must be taken when combining the two different types of breakpoint, as the results can be unexpected.

To see this, select file 3 and put a breakpoint on line 57 (which is in function test\_14). Then continue the program to reach that line. The commands to do this are:

```

file 3
breakpoint 57
continue

```

The result will look something like:

```

Catalina Command Line - blackbox debug_example

> update patricia.name[3] = '\0'
= 0x00

> print patricia
{
  name = "pat"
  age = 14 (0x000E)
  height = 1.90000
  spouse = 0 (0x00000000)
  mother = 507176 (0x0007BD28)
  father = 507232 (0x0007BD60)
  children = 0 (0x00000000)
  child_1 = 1583429642 (0x5E61380A)
  child_2 = 2454176181 (0x9247C1B5)
}

> file 3
list file = 3 : C:\Program Files (x86)\Catalina\demos\debug\debug_functions_2.c
list line = 1

debug file = 1 : C:\Program Files (x86)\Catalina\demos\debug\debug_example.c
debug line = 147

> breakpoint 57
* 00057 for (i = 0; i < 3; i++) {
> cont
*@ 00057 for (i = 0; i < 3; i++) {
>

```

You may expect that you can now use the **Step Out** command to go back to line 178 in **main** (which is the line after **test\_14** was called). However, this is not the case. Try it if you like – enter the command **out**. The program will instead step to the first line of **test\_15**.

To understand why this is so (and why BlackBox is a bit 'unorthodox' in this respect) you need to understand what BlackBox is doing with virtual breakpoints. The rules that apply are as follows:

- Step Next** adds a virtual breakpoint to every line in the current function and removes virtual breakpoints from the first line of every function in the program.
- Step Into** adds a virtual breakpoint to the first line of every function in the program.
- Step Out** removes the virtual breakpoints from every line in the current function, and adds a breakpoint to the first line of every function.

This logic means that everything works as expected when using *only* virtual or *only* user breakpoints – but when the two are used in the same program, it is possible to enter a function *without* having set the virtual breakpoints that mean BlackBox will return to the calling function (i.e. the virtual breakpoints on each line of the calling function). In summary:

1. **Step Next** will always work as expected, even after a user breakpoint.
2. **Step Into** will not work as expected after a user breakpoint. If the current function does not have virtual breakpoints on each line, use **Step Next** before using **Step Into**.
3. **Step Out** will not work as expected after user breakpoint. Instead, put a manual breakpoint after the calling instead of relying on **Step Out**. Or use **Step External** or **Step Up**.

## What to do next?

Some of the commands we have used in this tutorial have other options and uses. There are also other BlackBox commands we have not used – you can list them in BlackBox by using the **Help** command (note that the results may scroll off the screen):

**help**

The **BlackBox Reference Manual** contains full details on all BlackBox commands.

You can try out the other commands using the example program, or use BlackBox on any of the other demo programs provided with Catalina (after compiling them with the **-g** or **-g3** options).

## Known Issues with BlackBox

This release of BlackBox has a few known limitations:

You cannot use the Catalina Optimizer with BlackBox.

BlackBox cannot be used to debug multi-threaded programs.

BlackBox cannot resolve symbols that have been defined using **#define** – this can be confusing because in a source listing such symbols often look like any other variables or functions.

BlackBox does not fully implement absolute and relative addresses – addresses are sometimes displayed as absolute, and sometimes as relative. For example, displaying the address of a function in the debugger does not give the same result as taking the address of a function in the C program, assigning that address to a variable and displaying the variable in the debugger – this is because one is absolute and one is relative. Converting between the two requires knowledge of the current memory model in use by the program, the current relocation value and the current code segment offset.

In some circumstances, Linux may raise DTR when a serial port is opened – this may result in the Propeller that you are attempting to debug receiving a RESET signal whenever you start BlackBox. This problem does not occur when using the mouse port for debugging since that port has no connection to the Propeller reset line – but when using the normal serial port it may be necessary to temporarily disconnect the RESET line when using BlackBox. Some platforms provide a jumper for this purpose. On other platforms this must be done by disconnecting the RES line on the Prop Plug before starting BlackBox. Note that this is not an issue under Windows, which does not automatically raise DTR when a port is opened.