A Tutorial on using the BlackCat Debugger for Catalina

Background

This document provides some assistance to get you up and running with the BlackCat source level debugger for the Catalina C compiler.

It starts out with some basic information, but also covers some of the more advanced concepts – however, it is not necessary to complete reading the whole document to begin using BlackCat – in fact it is intended that you have BlackCat running as you work your way through the included example program.

What is BlackCat?

BlackCat is a Windows application that allows source level debugging of C programs that have been compiled using the Catalina C compiler, and are executing on a supported Propeller platform.

BlackCat runs on a host PC, and uses a serial link to communicate with the program executing on the Propeller.

BlackCat provides the ability to display C source files, set and clear user breakpoints, single step through C programs, and display or modify the values of the local or global C variables visible whenever the program is stopped.

What is the status of BlackCat?

BlackCat is currently in alpha release – it is being submitted to a selected group of alpha testers for initial functional testing and to elicit suggestions for improvement.

BlackCat is already capable of debugging arbitrary C programs, including all the example programs provided with Catalina.

BlackCat supports LMM programs on all platforms supported by Catalina, and LARGE mode XMM programs on some of those platforms – see the prerequisites section below.

SMALL mode XMM has also been implemented but not tested.

Some C language features have not been implemented.

What are the prerequisites for BlackCat?

The alpha release of BlackCat runs only under Windows. It has been tested under Windows XP SP2 (32 bit), but should also run on other versions of Windows.

Programs to be debugged using BlackCat must be compiled using Catalina 2.4 or greater – BlackCat will not work with any earlier versions of Catalina. Programs must be prepared by compiling them using a new **-g** command line option to Catalina (the meaning and use of this option is described later in this document).

BlackCat requires that the program being debugged has a cog free to manage the BlackCat serial link with the host PC. Other than that, the overhead of BlackCat 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 BlackCat – this makes BlackCat suitable for debugging real-time or time-critical programs.

Catalina itself can be used under either Linux or Windows – i.e. programs to be debugged using BlackCat under Windows can be compiled using Catalina under Linux. However, for the sake of brevity, this document assumes you are using both BlackCat and Catalina under Windows.

BlackCat also requires a USB or RS232 serial connection to the Propeller running the program to be debugged. The reason BlackCat does not support XMM on all platforms supported by Catalina is that BlackCat currently assumes this serial link is implement on pins 30 and 31, and this is not available on platforms that use the HX512 SRAM expansion card (e.g. the Hydra and the Hybrid). [1]

How do I install BlackCat?

You should completely install Catalina 2.4 before installing BlackCat. This document assumes you have installed Catalina in the default location (under Windows this is **C:\Program Files\Catalina**).

If you have not installed Catalina yet, see the **Catalina Reference Manual** for instructions. You should install at least the binary release of Catalina, plus the Catalina demo programs.

If you have already installed Catalina, but to a directory other than the default directory, then all the Catalina commands and options shown will be identical – but some of the directory names will need to be changed.

BlackCat can be installed by unzipping the distribution provided in the main Catalina directory (e.g. **C:\Program Files\Catalina)**. This will replace a few of the normal Catalina files, and add some new ones specific to BlackCat.

If you are using **Code::Blocks**, BlackCat includes a replacement User Template for Catalina that will build both a **Release** and a **Debug** version of all programs. Install the new **Catalina.cdb** provided in the BlackCat distribution over the top of the existing template – this will be in a location such as:

```
C:\Documents and Settings\<your name>\Application Data\codeblocks\UserTemplates\Catalina
```

Why is BlackCat "unorthodox"?

The BlackCat debugger uses a slightly unusual approach to breakpoints, which are of two types – user or virtual. This can lead to unexpected results when a program is executed using a combination of the user breakpoint operations (set and clear breakpoint, run to breakpoint), and the various virtual breakpoint operations (step next, step into, step out).

This is more fully explained later in this document.

Does BlackCat supersede POD?

Not entirely. BlackCat is much easier to use for debugging C programs, and it also supports XMM programs (which POD does not). BlackCat is therefore expected to become the preferred "Catalina users" tool, but POD will continue to be included with Catalina, as a "Catalina gurus" tool.

[1] Plans are underway for the next release of BlackCat to support arbitrary pins for serial communications, allowing platforms that use the HX512 to be supported by the use of additional hardware to implement a serial link on other pins.

Being an assembly language debugger, POD is useful for debugging the assembly code generated by Catalina, or debugging any changes to the Catalina LMM kernel. In fact, POD was used to debug the code added to the kernel to support BlackCat!

Who developed BlackCat?

BlackCat was developed co-operatively by Bob Anderson and Ross Higson.

Bob Anderson is responsible for the overall design of BlackCat, the parsing of debug information generated by Catalina into a form suitable for use with BlackCat, the BlackCat user interface and the BlackCat debug cog that interacts with the Catalina Kernel.

Ross Higson is responsible for Catalina. Changes required to Catalina to support BlackCat include a modified Catalina code generator that emits debugging information during compilation, BlackCat specific targets that incorporate BlackCat-specific program initialization code, and BlackCat support specific to each Catalina LMM and XMM Kernel.

Are there any known problems with BlackCat?

Yes, there are a few things not yet implemented, and a few other issues – see the section **Known Issues** later in this document.

Who should I contact about BlackCat?

Please contact Ross Higson at ross@thevastydeep.com

Why is it called BlackCat?

During World War II, a Catalina used for combat missions – particularly night missions – was often painted matt black and known as a “Black Cat”.

See http://www.daveswarbirds.com/blackcat

Preparing programs for use with BlackCat

Overview

Preparing a program can be as simple as including the **–g** command line option when compiling the program with Catalina. This instructs Catalina to do two things:

1. Generate additional information that is required by the BlackCat 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 BlackCat.
2. Select a BlackCat enabled target for the program. BlackCat targets are provided for LMM, EMM and XMM programs. These targets contain additional initialization code that loads the BlackCat debug cog, and performs other additional tasks required to prepare the kernel for co-operation with the debug cog.

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

- The program must have a spare cog available.

- The program must allow Propeller pins 30 and 31 to be used for serial communication with the BlackCat debugger running on the host.
- You cannot specify your own target – to use BlackCat you must let Catalina select a BlackCat-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, so the final overhead is only about 400 bytes.

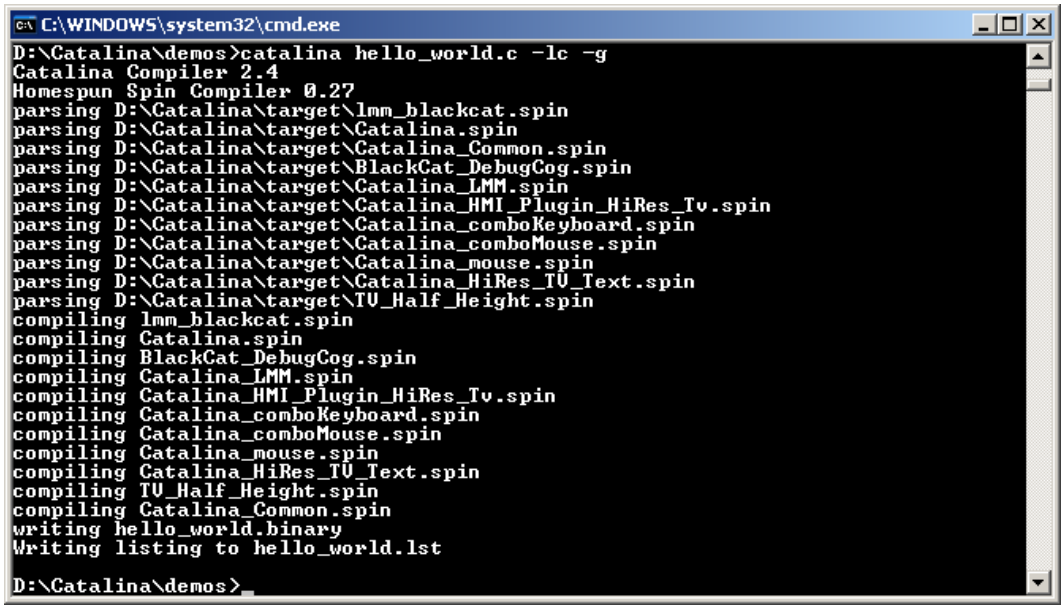
Preparing a program using Catalina from the command line

As mentioned above, simply include the **–g** command line option along with all the other normal options when compiling the program.

For example, after setting up to use Catalina (via the **use_catalina** batch file) execute the following command in the Catalina demo directory:

```
catalina hello_world.c -lc -g
```

This will produce output similar to the following:



Note that Catalina has automatically used the **lmm_blackcat.spin** target (instead of the more usual **lmm_default.spin** target).

In addition to the normal **hello_world.binary**, Catalina will also generate the following files:

- A file called **hello_world.lst** (a listing file is *always* produced when the **–g** command-line option is included).
- 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).

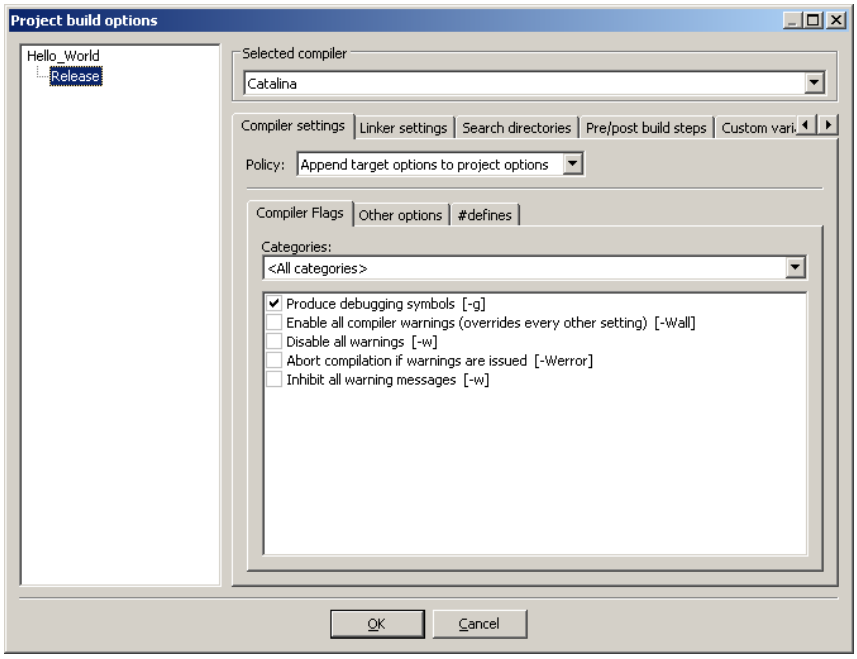
Preparing a program using Catalina from Code::Blocks

BlackCat provides an updated version of the **Code::Blocks** Catalina User Template to be used for creating new Catalina projects. This template will automatically set up new projects to build either a **Debug** or a **Release** version of your program. The Debug version will have the **–g** option enabled, while the Release version will not.

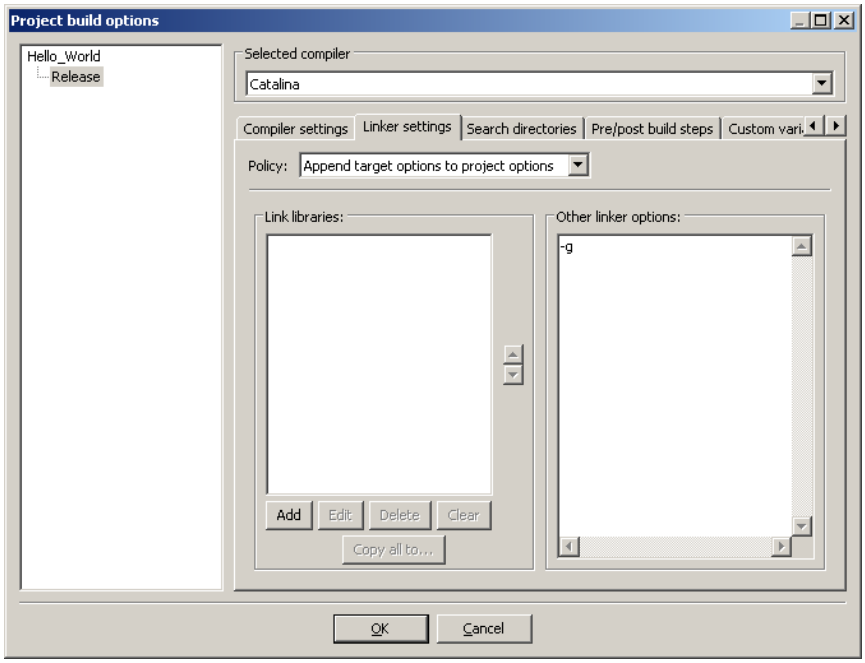
To select which version to build, use the menu command **Build->Select a Target**. The Release version will be built in a **bin\release** subdirectory and the debug version will be built a **bin\debug** subdirectory of the main project directory.

However, for existing projects you will have to tell **Code::Blocks** specifically how to build a debug version of your program – one way to do this is by modifying the Release version to specify the **–g** option in two places (both accessible via the **Project -> Build Options** menu item):

1. On the Compiler Settings tab, check the **Produce Debugging Symbols [-g]** Compiler Flag:



2. On the Linker Settings tab, check the **-g** to the set of **Other linker options**:



Then you simply compile and link your program as normal. Alternatively, a second **Debug** target could be added to the project alongside the **Release** target, which will then need to have the **–g** flag added (as described above).

Note that the **.debug** files generated by Catalina will be placed in the same directory as the source files, but the **.lst** and **.dbg** files will be placed in the same directory as the binary output file. This is important to know because you will have to load the dbg file into BlackCat (this is described later).

Loading a program into the Propeller

Neither Catalina nor BlackCat currently provide any support for loading programs into the Propeller[2]. You will have to use one of the following methods:

1. For LMM programs, you can load them using the Parallax Propeller tool.
2. For EMM programs, you can program them into an EEPROM and then reboot the Propeller.
3. For LMM or XMM programs, you can write them to an SD card and then use the Catalina Generic SD Program Loader. For multi-CPU systems such as the TriBladeProp or Morpheus, you may also need to use the Catalina Generic SIO Program Loader.

In any case, all programs built using the **–g** flag always stop after the execution of the initialization code, and then wait till the BlackCat debugger establishes serial communications with the debug cog.

[2] This will be rectified in a later release of Catalina and/or BlackCat.

Starting BlackCat

To start BlackCat, simply enter the following command:

```
blackcat
```

This command accepts no parameters. See the next section on how to configure and use BlackCat once it has started.

Using BlackCat

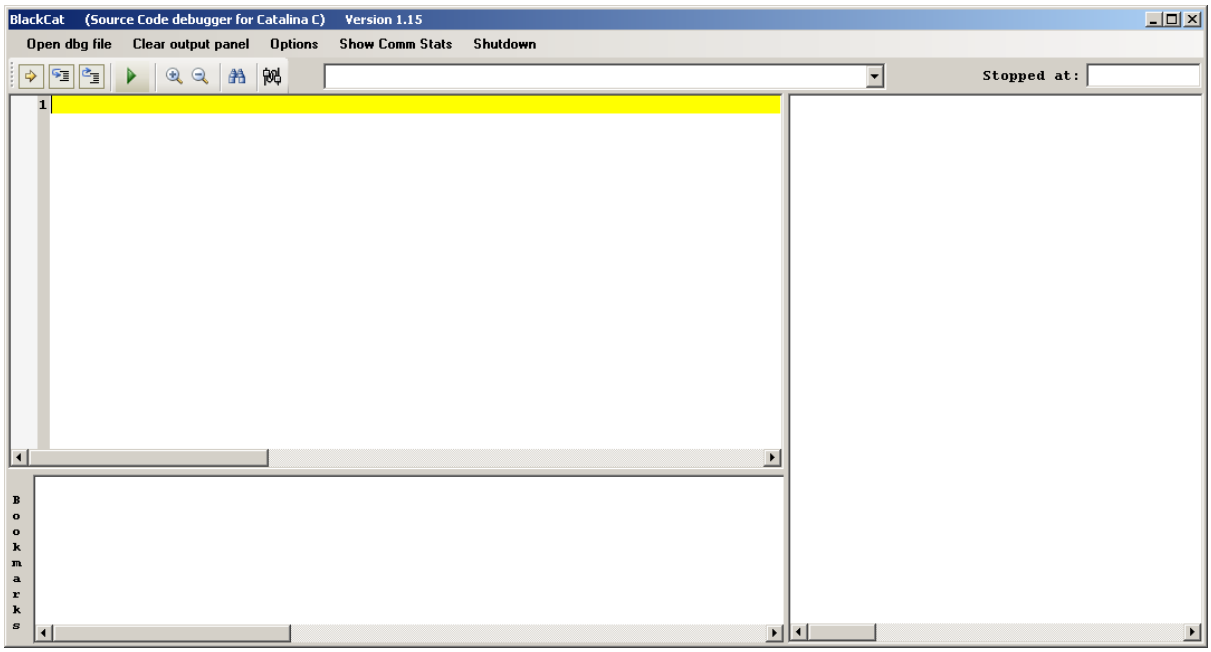
Overview

This section contains a general introduction to each component of BlackCat, such as the windows, menus and various commands.

The BlackCat Window

When BlackCat is first started, it opens two windows. The larger window is referred to as the BlackCat window. This window is used for displaying source files, setting and clearing breakpoints, viewing and modifying program variables.

It will appear similar to that shown below:



The BlackCat Window contains:

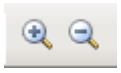
- A menu bar for executing commands
- A tool bar with buttons, a file list dropdown box, and a status indictor. The meaning of the buttons is as follows (their use is described in more detail later):


Step (to the next virtual breakpoint);


Step Into (any procedure called on the current line);


Step Out (back to the calling function);


Run (to next user breakpoint);


Increase (left click) or decrease (right click) the font size;


Search for text in source files;


Open (or bring to top) the Propeller Communications window.

The file list dropdown box will list all the C source files used in the current program, and is used to select which source file to display.

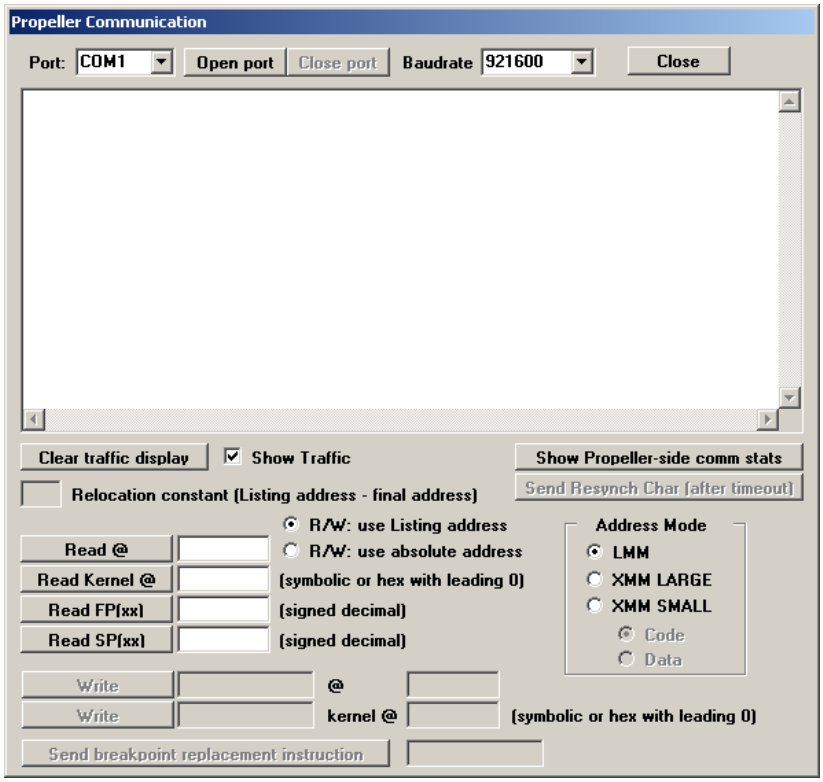
The status indicator that indicates whether the program is running or stopped, and the current program counter.

- A Source panel. This is used to display the file selected in the file list dropdown box.
- A Bookmark panel. This is used to display a list of lines that have been “marked” for future attention. It allows easy navigation back to the line in the future.
- An Output panel. This is where all status information is displayed, and where the variables visible at each breakpoint are listed.

The Propeller Communications Window

The smaller window that opens when BlackCat is started is referred to as the Propeller Communications window.

It will appear similar to that shown below:



This window contains controls for:

- Selecting the serial port to use, and the baud rate, and opening or closing the serial port.
- Closing the Propeller Communications Window (it can be re-opened at any time from the toolbar of the main BlackCat window).
- Viewing communications traffic between BlackCat and the Propeller program.
- Displaying Kernel and Hub or XXM RAM values.
- Selecting whether to use Listing or Absolute addresses. Listing addresses are those used in the leftmost column of the listing file produced by Catalina – but you may notice that these are not the actual addresses used within the instructions themselves – those are absolute addresses.
- Selecting the Address Mode: LMM, XMM SMALL or XMM LARGE. It is important to do this before a dbg file is loaded.
- Modifying Kernel and Hub or XXM RAM values (these controls are disabled by default).

Connecting to the Propeller

The first thing to do once BlackCat is started is to select the serial port and baud rate. Both USB and RS232 ports can be used. A baud rate of 115200 should be specified (the program defaults to this). While the baud rate can go up to 921600 but this requires changes to be made to the compiled program (i.e. by manually editing **Catalina_Common_Input.spin**).

A baud rate of 115200 was chosen as the default as it is appropriate for both USB and RS232 ports. Higher speeds may work on some platforms but not others.

Once these are selected, press the **Open port** button. An error message will be displayed if the port cannot be opened. If the port is opened successfully, there is no message – but you can check if BlackCat is able to communicate with the debug cog running on the Propeller by pressing the **Show Propeller-side Comms Stats** button.

Once the **Com port** and **Baud Rate** are selected, they are remembered from sessions to session.

Select the Address Mode

Before loading any dbg file, it is important to check the **Address Mode**. Eventually, this may be selected for you automatically whenever a dbg file is loaded, but at present it must be done manually ***before loading the dbg file!***

If you fail to select the correct Address Mode, the results will be unpredictable, and you will need to shutdown BlackCat and also reload the binary program into the Propeller.

The setting for this is also remembered from session to session – but it is wise to check it each time.

BlackCat Menu Bar Commands

Open dbg file

You need to tell BlackCat to load a dbg file, which must match the binary currently being executed on the Propeller. If the two do not match, the results will be unpredictable.

Since BlackCat immediately starts setting up Virtual Breakpoints once a dbg file is loaded, the Propeller Communications Window must be used to configure and open the serial port, and select the address mode, before a dbg file is loaded. This is why the Propeller Communications window is always opened on top of the BlackCat window. However, once this task is done the Propeller Communications Window can often be closed for the remainder of the debug session.

Before a dbg file is loaded, about all you can do is read specific locations from the kernel or from memory using an absolute address – using listing addresses instead of absolute addresses is not be valid because it requires information in the dbg file (this information is displayed in the **Relocation Constant (Listing address – final address)** field once a dbg file has been loaded.

Clear Output Panel

This command simply clears the contents of the output panel.

Options

The options menu contains the following additional commands:

Show Virtual Breakpoints

This option (selected by default) determines whether virtual breakpoints will be shown in the left margin of the source panel. This is handy as it tells you those lines that have had code generated for them (and hence can hav a breakpoint added) and those that didn't.

Show Traffic in Communications Window

This option causes messages to be logged for all commands between BlackCat and the debug cog visible in the Propeller Communications window. It overrides the Show Traffic option in that window (which does a similar thing).

Remove all Breakpoints

Remove all user and virtual breakpoints that BlackCat has inserted into the program.

Show Comms Stats

Print current BlackCat communications statistics in the output window. Clear the counters.

Shutdown

Remove all virtual and user breakpoints (if possible), and let the program run as normal.

This is only possible if the program is still under the control of BlackCat – i.e. it is stopped at a breakpoint.

Displaying files in the Source Panel

Once a dbg file is loaded, the source file dropdown list will be populated with the names of all the C source files used in the compilation. The source file containing the **main** function will be displayed initially, and the program will be stopped at the entry point to **main**.

The left hand column of the source panel shows whether or not there are any virtual or user breakpoints that refer to the line, and also whether the program is stopped on the displayed line:


Virtual Breakpoint


User Breakpoint


Program stopped at Virtual Breakpoint


Program stopped at User Breakpoint

Right clicking in this window brings up the following menu:

Bookmark highlighted line adds the highlighted line to the lines listed in the bookmark panel. **Remove all breakpoints from all files** is similar to the **Remove all Breakpoints** menu item.

Using Bookmarks

Right-clicking the mouse on a line in the bookmark panel brings up the following menu:

Go to { line } … displays the highlighted line in the source panel. **Delete highlighted bookmark** removes a bookmark, and **Clear bookmarks** removes all bookmarks from the current file.

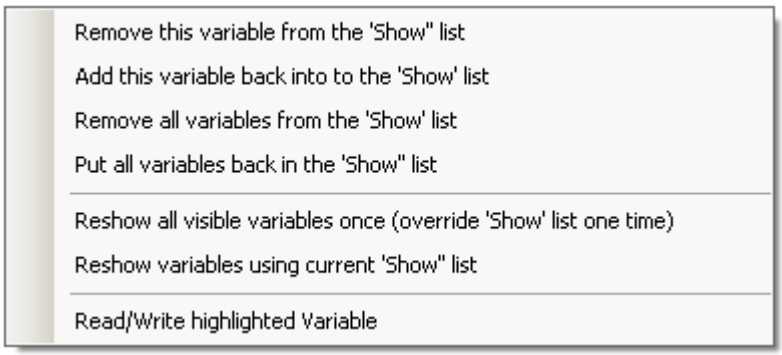
Only the bookmarks for the current file are shown. Bookmarks are saved from session to session.

Managing the Variable "Show" list or Read/Write Variables

The "Show" list is the list of all variables – global or local – that are visible from the breakpoint at which BlackCat is currently stopped. It is displayed in the output panel after each and every breakpoint (user or virtual).

If the list of such variables is large, this can be slow, and make it difficult to locate a value of particular interest.

So BlackCat provides a means of managing the "Show" list. Right clicking on a highlighted line in the output panel brings up the following menu:



Remove this variable from the 'Show' List – remove the highlighted variable from the list of those shown at each breakpoint.

Add his variable back into the 'Show' list – show the highlighted variable at each breakpoint.

Remove all variables from the 'Show' list – do not show any variables at each breakpoint.

Put all variables back in the 'Show' list – show all variables at each breakpoint again.

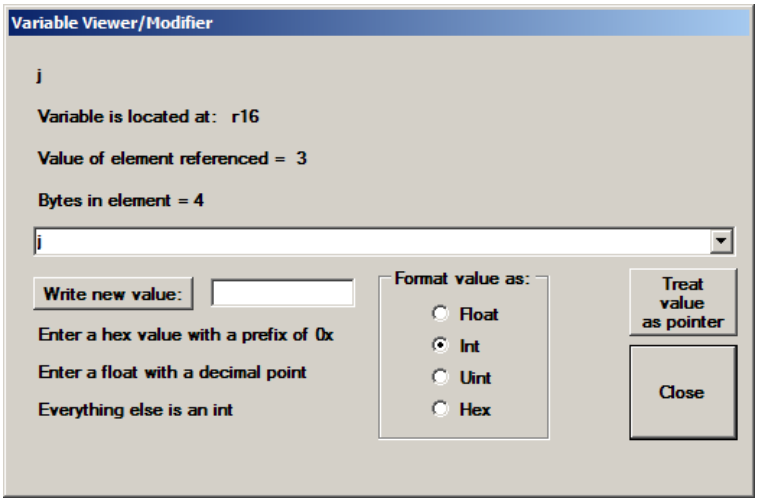
Reshow all visible variables once (override 'Show' list one time) – show all variables once – useful to find variables that may need to be added back into the show list.

Reshow variables using current 'show' list – redisplay the current list – useful if there are variables that may be updated by another executing cog.

Read/Write highlighted Variable – open a dialog box that allows the value of the highlighted variable to be displayed in more detail, and/or modified (see next section).

Displaying and Editing Variables

When the “Show” list is displayed in the output window, highlighting a line containing a variable (by left clicking on it) and then right clicking to bring up the menu and selecting Read/Write highlighted variable displays the **Variable Viewer/Modifier** dialog box:



This box contains the name of the variable, its location (register, frame or global), its current value, and its size (typically 1, 2 or 4 bytes for simple scalars). Also, if the variable is not a simple scalar (e.g. it is a structure, union or array) the dropdown list will enumerate all the possible scalars that comprise the variable.

A new value can be written to each scalar, or scalar component of a more complex variable.

If the variable contains a pointer it can be de-referenced to display the value that the pointer currently points to (and de-referenced again, if that value is also a pointer).

The format used to display scalar variables is selectable.

An example of using BlackCat

Overview

This section provides a real-world example of using BlackCat. It uses an example program provided with the BlackCat distribution, which will be located in the **blackcat_demo** subdirectory once BlackCat has been installed.

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

Compiling the example program

The example program provided consists of the following C source files:

```
debug_test.h
debug_main.c
debug_functions_1.c
debug_functions_2.c
```

To build the example program, go to the demo director and run the **build_example.bat** batch file provided, specifying your Propeller platform as a command line parameter. If that platform has multiple CPUs, also specify the CPU.

For example:

```
build_example TRIBLADEPROP CPU_1
```

or

```
build_example DRACBLADE
```

This will generate the following files:

```
example.binary
example.dbg
example.lst
debug_main.debug
debug_functions1.debug
debug_functions2.debug
```

A **Code::Blocks** project is also provided, which by default builds the example for the **TRIBLADEPROP CPU_1** platform – you may need to edit this to suit your platform.

Note that if you use the batch file to compile the example, the binary output file (and corresponding lst and dbg files) will be located in the **blackcat_demos** subdirectory, whereas if you use **Code::Blocks** to compile the example, the files will be located in the **blackcat_demos\Example\bin\Debug** subdirectory.

Loading the example program

The example program is compiled as an LMM program, so the binary output file **example.binary** can be loaded into the Propeller using the Parallax Propeller Tool.

When the binary has been loaded, all the drivers are initialized, so you should see a blank display appear. However, the C program will not begin to execute until we start BlackCat.

Starting BlackCat, establishing comms and loading the dbg file

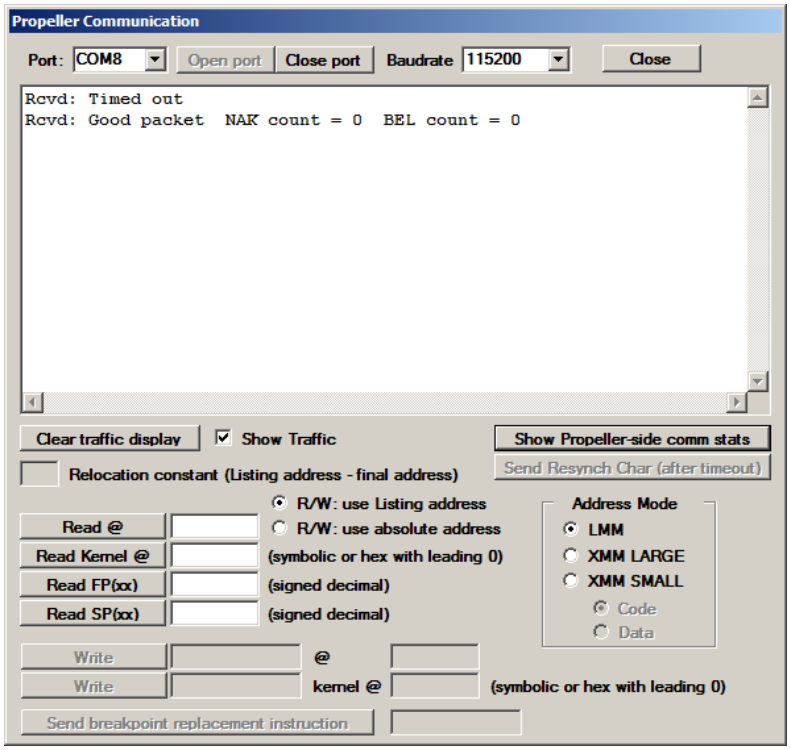
Start BlackCat using the command:

```
blackcat
```

When the BlackCat and Propeller Communications windows open, select the serial port and baud rate (use 115200 baud). Then press **Open port**.

There is no message, but you can manually verify the serial communications is working by pressing **Show Propeller-side comm stats**. You may get an error on the first attempt, but the second press should work.

Here is what you might see if you had your Propeller connected to COM8:

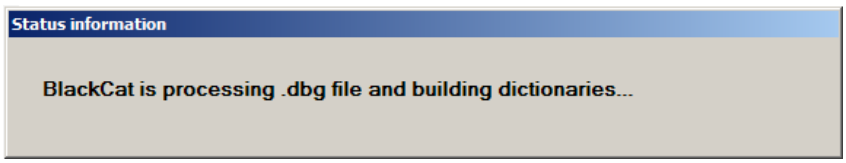


You can now close the Propeller Communications window (press the **Close** button at the top right).

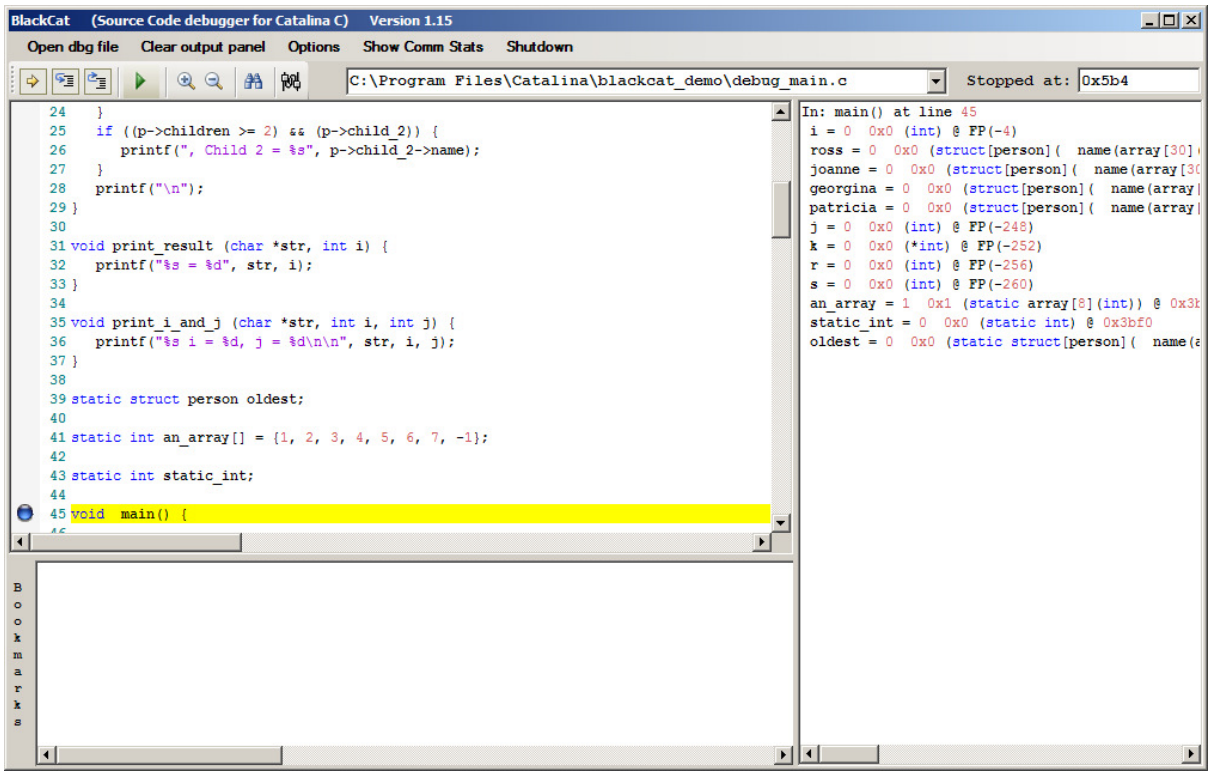
The Propeller Communications window can be re-opened by pressing the  button on the toolbar of the main BlackCat window.

In the BlackCat main window, Press **Open dbg file**. A dialog box will appear in which you can select the dbg file to load – it must be one that corresponds to the binary you have loaded into the Propeller.

A message will appear similar to the following while the dbg file is being loaded and processed:



One the load is complete (it may take a few seconds) the example program should appear in the BlackCat source panel, similar to that shown below:



Note that the blue circle in the margin means the program is stopped at a virtual breakpoint – this virtual breakpoint is automatically added by BlackCat on initialization.

Note also that the progress indicator (top right) indicates the address the program is stopped (in this case **0x5b4**), and the output panel contains a message about where the program is stopped (**in main() at line 45**) and also a printout of all the variables visible at this point.

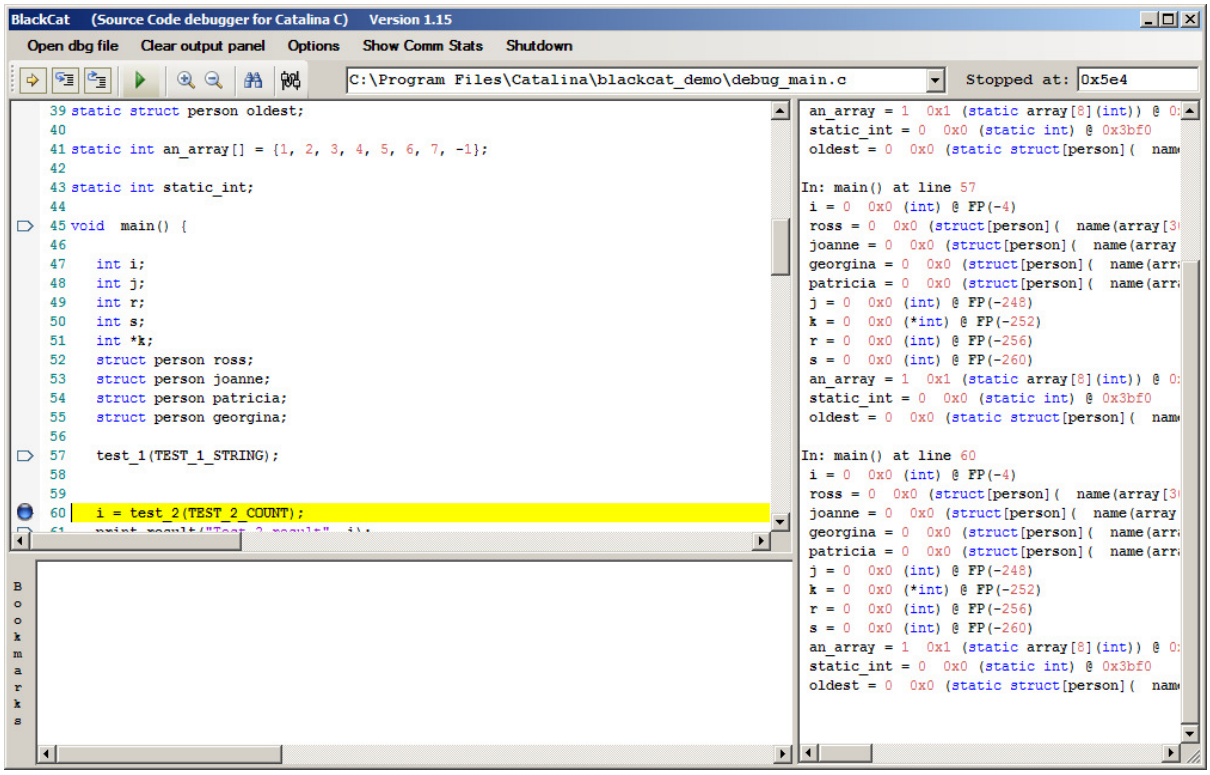
Scroll within the source panel to display more of the main function. You will notice markers for virtual breakpoints that have been set at every executable line in the **main** function. These virtual breakpoints are used to step through the program line by line, and show where the program will stop at each step (since not all lines have code generated for them).

Using Step, Step Into & Step Out

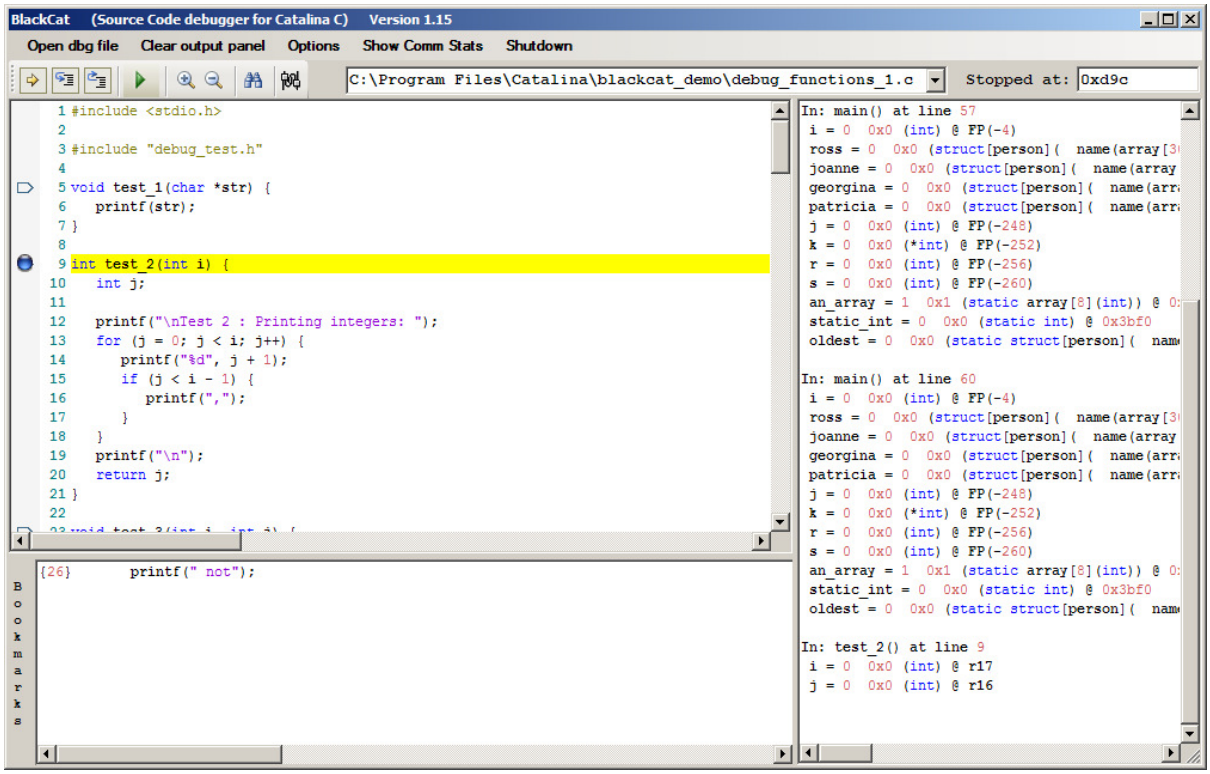
To execute a single ‘Step’, you press the  button once. Do this now. The program should step to line 57, the variable list will be displayed in the output window, and the source panel will appear.

Press ‘Step’ one more time to step to line 60. The display attached to the Propeller should now show a line of text on it (put there by the function call executed on line 57).

The BlackCat window should now appear similar to the following:



This time, instead of pressing ‘Step’ (which steps over function calls), press  to ‘Step Into’ those calls. The program will step into the function **test_2**.



Now press ‘Step’  to step to line 12. Notice how the virtual breakpoints change as you do so.

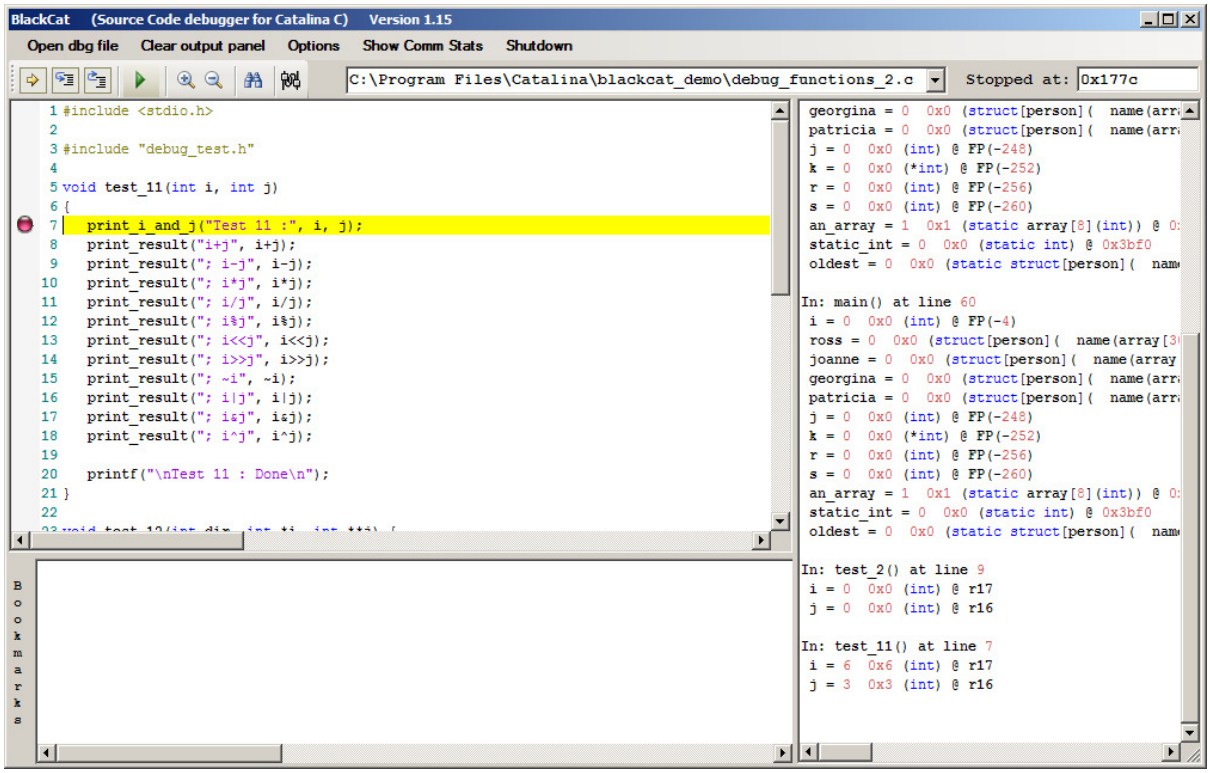
Then press 'Step Out'  to return to line 61 – i.e. the line after the procedure we 'Stepped Into'.

User Breakpoints

Sometimes, we do not want to single step through the program – we just want to set a user breakpoint on a specific line and let the program run till it reaches it.

To demonstrate user breakpoints, we will set a breakpoint on line 7 of the file **debug_functions_2.c** (in function **test_11**). First, change file by selecting the file from the file drop down list at the top of the file window. Then put a user breakpoint by clicking next to line 7 in the left hand margin.

The result will look something like this:



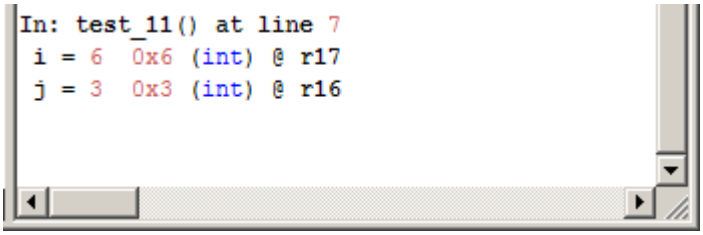
Now we simply want to let the program continue till it reaches this user breakpoint.

To do that we use the 'Run'  button. This button will cause the program to execute till it reaches the user breakpoint – it will not stop at virtual breakpoints on the way. Press it now.

There may be a pause while the program removes all the virtual breakpoints it has previously placed (to support the single stepping we did earlier), then the program will proceed.

Note that the program is interactive – you need to press enter on the keyboard a couple of times. Then the program will stop at line 7.

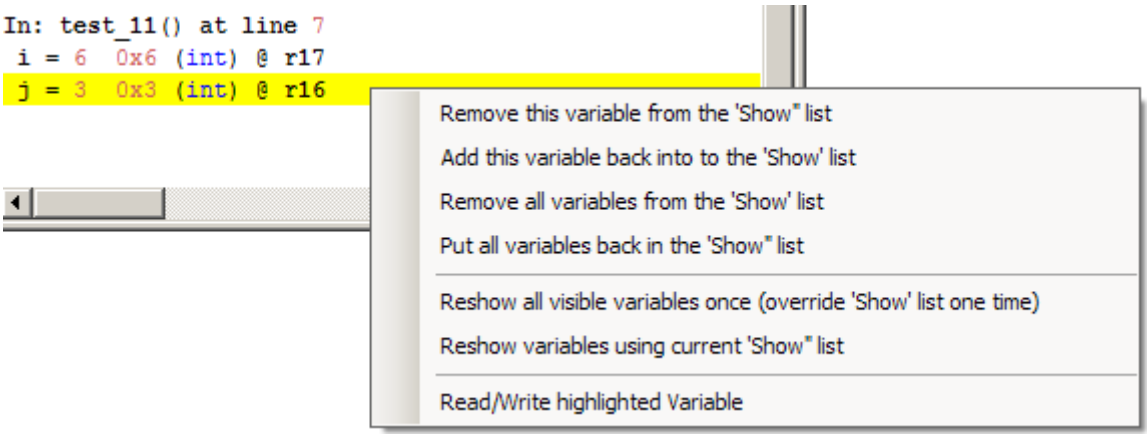
Note that the output panel displays the values of the variables **i** and **j**:



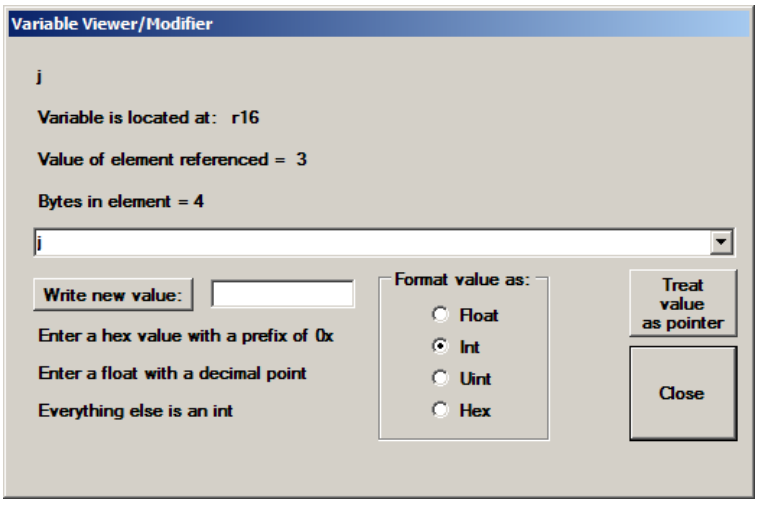
We will now modify one of these variables and then let the program continue.

Viewing and Modifying Variables

To view or modify a variable, left click to highlight the line containing the variable in the output panel (in yellow), then right click to bring up the menu and select **Read/Write highlighted Variable**.



A dialog box similar to the following will appear:



This dialog box can be used to display the scalar value of the variable (as a **float**, an **int**, an **unsigned int** or a **hex** value), or to de-reference it (if the variable in fact contains a pointer to a scalar data type rather than the data type itself) by pressing the **Treat value as pointer button**. It can also be used to change the current value of the variable.

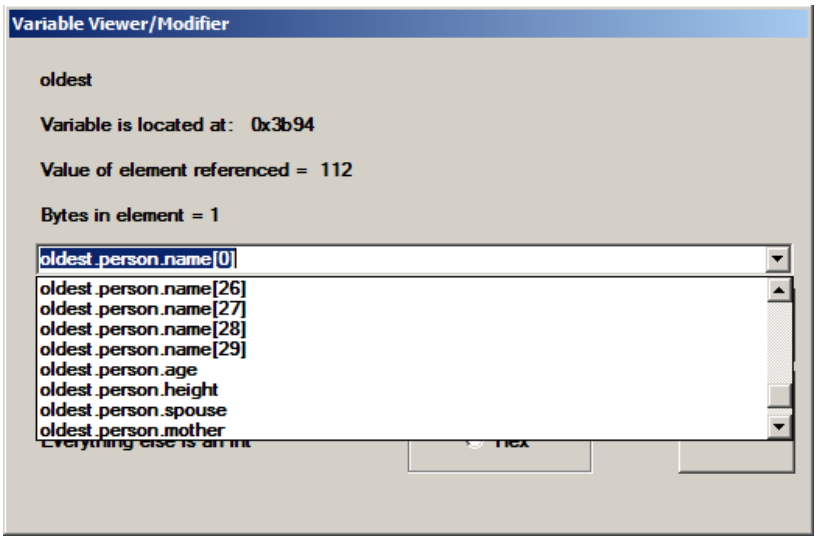
You can change the value by typing the new value in the field next to the **Write new Value** button, then pressing that button, then **Close** to close the dialog box. Try this now – change the value of **j** to 4.

If we 'Step' the program, we will see the new value of **j** in use.

The dropdown list is used to allow the selection of the element of an array, or the member of a structure. For a simple scalar variable (such as **j**), the dropdown list contains only the name of the scalar variable, and can be ignored. To see it in use for a non-scalar variable, remove the user breakpoint from line 7 of the current file, and select the file **debug_main.c.** Put a user breakpoint on line 132 in function **main**, just after the calls to **test_11**. Then press 'Run' (if you didn't remove the user breakpoint in function **test_11**, you will need to press 'Run' more than once since that function is called twice).

Then highlight the line containing the variable **oldest** and open the Read/Modify dialog box again.

This time, the dropdown list contains a list of all the elements of the **oldest** structure, including the element **name**, which is an array:



In addition to selecting from the list, you can also type in the name of the structure member or array element and press TAB. If the name is valid it will be displayed.

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 the file **debug_functions_2.c** again and put a breakpoint on line 41 (in function **test_13**). Then press 'Run' to reach that line.

You may expect that you can now use the 'Step Out' operation to go back to line 144 of **debug_main.c** (i.e. the line after **test_13** was called). However, ***this will not work***. Try it if you like – you will simply lose control of the program, which will then run to completion without stopping at any further breakpoints.

To understand why this is so (and why BlackCat is a bit 'unorthodox' in this respect) you need to understand what BlackCat is doing with virtual breakpoints.

When main is first entered, a virtual breakpoint is placed on each line of the **main** function. Then the rules that apply are as follows:

'Step' *adds* a virtual breakpoint to every line in the *current* function only (if not already present).

'Step Into' *adds* a virtual breakpoint to the first line of *every* function in the program (if not already present).

'Step Out' *removes* the virtual breakpoints from every line in the current function only, and from the first line of every function.

'Run' *removes* all virtual breakpoints so the program will only stop on user breakpoints.

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 BlackCat relies on to enable it step out of the function again (i.e. the virtual breakpoints on each line of the calling function). In summary:

1. 'Step' 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' 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'. Once back in the calling function.

Known Issues with BlackCat

This alpha release of BlackCat has a few limitations and known issues – most of these are with the **Variable Viewer**/**Modifier** dialog box:

- Bit fields are not implemented;
- Enums are not implemented;
- Unions and structures embedded within a structure are not implemented.
- Arrays of pointers to structures cause an exception;
- There is no specific char or char array types – use **int** or **hex** to view individual characters instead.
- The format is not automatically selected (e.g. to display a float value you have to manually specify that the variable is in fact a **float** and not an **int**).

Other miscellaneous issues:

- SMALL XMM mode has been implemented but not tested.
- The source panel cannot be used to display **.h** files.
- BlackCat cannot resolve symbols that have been redefined using **#define**.
- The length of the file selection dropdown list is too short too display the full filename in some cases.

Advanced Topics

Debug files and listing files

The **.debug** files contain 'stabs' format information (a common format used to hold symbol table information for use by debuggers) as generated by the LCC compiler. This is the raw information that is used to build the **.dbg** file that BlackCat uses.

The listing files are generated by Homespun, and are useful for determining program code and data addresses for use in BlackCat. However, note that the addresses given on the listing are not "real" addresses – they primarily represent where the object ended up in the binary file generated by Homespun, and this is not necessarily where the object will be physically loaded at run time. However, once a dbg file has been loaded into BlackCat, BlackCat can do the necessary conversion to allow either listing or absolute addresses to be used.

Modifying Kernel and Hub / XMM memory locations

To enable the normally disabled functions in the Propeller Communications Window to allow kernel and hub memory locations to be modified directly, enter the word **toggle** in the **Read Kernel @** data entry box, and then press the **Read Kernel @** button.

Note that this can only be done while BlackCat is stopped at a breakpoint.

Modifying or Recompiling programs

To prevent erroneous results, if BlackCat detects that important files such as the *source* files or the *dbg* files are modified while the program is being debugged, BlackCat will print a message in the output panel telling you to shut BlackCat down and restart it. Not doing so when this message appears will lead to unpredictable results.

