Blink User Guide

A debugger contributed to xtc, Version 1.xx.x (07/07/08)

Byeongcheol Lee, Martin Hirzel, Robert Grimm and Kathryn McKinley

The current Blink project members are Robert Grimm, Martin Hirzel, Byeoncheol "BK" Lee, and Kathryn McKinley.

UT Austin: This work was supported by CNS-0719966, NSF CCF-0429859, NSF EIA-0303609, DARPA F33615-03-C-4106, Intel, IBM, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

NYU and IBM: This material is based in part upon work supported by the National Science Foundation under Grants No. CNS-0448349 and CNS-0615129 and by the Defense Advanced Research Projects Agency under Contract No. NBCH30390004.

This is the user guide for a debugger contributed to xtc Version 1.xx.x (07/07/08).

Table of Contents

1 Introduction

Blink is a portable mixed-mode Java/native debugger for JNI (Java Native Interface) and Jeannie programs. Blink allows you to inspect Java and native program state while the JDB or the GDB shows only one of Java or the native program state.

Blink is portable across different Java virtual machines, machine architectures and operating systems. Blink combines an existing poratable Java debugger (JDB) and native debugger (GDB). These black-box sub-debuggers under the Blink control cooperate transparently with each other to create an illusion of a single mixed-mode Java/native debugger.

The Blink user interface is similiar to the command line interfaces in JDB and GDB. The Blink master script file (`blink.sh`) provides the same invocation command line syntax as JDB. The Blink commands within the debugging session are similar to these in JDB and the GDB. JDB and GDB users should be familiar with the Blink debugger. This user guide describes how to use the Blink mixed-mode Java/native debugger in practice.

1.1 Installation

This section describes how to install xtc, which includes the Blink debugger, and how to test that the installed Blink debugger runs correctly.

1.1.1 Requirements

The Blink debugger uses Java Standard Edition version 5 or higher, JDB 1.5 and GDB 6.7.1 and Linux-2.6.22 or higher under x86. We have tested Blink with IBM J9 1.5 and Sun Hotspot 1.6 running on the Linux-2.6.22. We plan to include more operating systems and architectures. For instance, we are actively working on using the Microsoft CDB and supporting the Windows native environment.

1.1.2 Download

You need xtc-core.zip to run Jeannie, xtc-testsuite.zip to test your local Jeannie installation, and antlr.jar and junit.jar to compile xtc. You can download these four files from their respective project websites, for example like this:

```
wget http://cs.nyu.edu/rgrimm/xtc/xtc-core.zip
wget http://cs.nyu.edu/rgrimm/xtc/xtc-testsuite.zip
wget http://www.antlr.org/download/antlrworks-1.1.4.jar
wget http://downloads.sourceforge.net/junit/junit-4.4.jar
```

Pick a directory where you want your local xtc and Jeannie installation to live. Assuming your directory is called `local_install_dir`, populate it with your downloads like this:

```
unzip -d local_install_dir xtc-core.zip
unzip -d local_install_dir xtc-testsuite.zip
mv antlrworks-1.1.4.jar local_install_dir/xtc/bin/antlr.jar
mv junit-4.4.jar local_install_dir/xtc/bin/junit.jar
```

1.1.3 Configuration

You need to set your `PATH` environment variable to include your Java 1.5 compiler and JVM. In addition, you need to set `PATH_SEP` either to ‘`:`’ on Linux or Mac OS X or to ‘`;`’ on Windows/Cygwin. Assuming you unzipped xtc to a directory called `local_install_dir`

and you installed JDK 1.5 or higher in `jdk_home`, you now need to perform the following steps:

```
export PATH_SEP=':'
export JAVA_DEV_ROOT=local_install_dir/xtc
export PATH=$JAVA_DEV_ROOT/src/xtc/lang/blink:$PATH
export CLASSPATH=$JAVA_DEV_ROOT/bin/junit.jar$PATH_SEP$CLASSPATH
export CLASSPATH=$JAVA_DEV_ROOT/bin/antlr.jar$PATH_SEP$CLASSPATH
export CLASSPATH=$JAVA_DEV_ROOT/classes$PATH_SEP$CLASSPATH
export JAVA_HOME=jdk_home
make -C $JAVA_DEV_ROOT classes configure
```

The last step will use `xtc/Makefile` to compile and configure xtc along with the Blink debugger. You may see some warning messages related to Java generics, but the compilation should keep going and finish without any fatal error messages.

1.1.4 Testing the installation

After completing the download and configuration step, try the following:

```
(bash) cd $JAVA_DEV_ROOT/fonda/blink_testsuite
(bash) make
...
```

Make sure that the JDB correctly runs three examples.

```
(bash)jdb PingPong
Initializing jdb ...
> run
run PingPong
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started:
jPing: 3
cPong: 2
jPing: 1
cPong: 0
The application exited

(bash)jdb CompoundData
Initializing jdb ...
> run
run CompoundData
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started:
2.5029
2.71828
3.14159
4.6692
The application exited

(bash)jdb JeannieMain
Initializing jdb ...
> run
```

```
run JeannieMain
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started:
1
3
2
The application exited
```

Make sure that the Blink master script correctly runs all the three examples.

```
(bash) blink.sh PingPong
Blink a Java/C mixed language debugger.
(bdb-j) run
jPing: 3
jPing: 1
cPong: 2
cPong: 0
(bash) blink.sh CompoundData
Blink a Java/C mixed language debugger.
(bdb-j) run
2.5029
2.71828
3.14159
4.6692
(bash) blink.sh JeannieMain
Blink a Java/C mixed language debugger.
(bdb-j) run
1
3
2
```

1.2 Trouble shooting

As with any complex piece of software, you may run into trouble when trying to use the Blink debugger. This section describes a few common issues and how to address them. We will keep updating this section as we encounter additional difficulties and their solutions.

If you cannot compile the Blink debugger at all, or if it does not run, you should double-check whether all the required tools are installed on your local machine. In particular, you need Java 1.5 or higher, and you need the GNU GDB, see Section 1.1.1 [Requirements], page 1. Next, try the tests that come with Blink, see Section 1.1.4 [Testing the installation], page 2. Finally, double-check that you set your environment variables correctly, in particular, `PATH`, `CLASSPATH`, and `LD_LIBRARY_PATH`.

If the Blink debugger throws an internal exception rather than producing a nice error message, that's a bug; please report it, along with a minimal test case that reproduces it.

2 Examples

This chapter discusses Blink through the use of examples. Each section uses a short self-contained piece of code to illustrate one aspect of how to use the Blink debugger.

The following `PingPong` example in the Blink test suite illustrates how to examine the transitions between Java code and native code with JNI. We show the code and then use it to illustrate how to use Blink.

```
(bash) cd  $JAVA_DEV_ROOT/fonda/blink_testsuite
(bash) cat -n PingPong.java
 1 public abstract class PingPong {
 2   static {System.loadLibrary("PingPong");}
 3   public static void main(String[] args) {
 4     jPing(3);
 5   }
 6   static int jPing(int i) {
 7     System.out.println("jPing: " + i );
 8     if (i > 0 )
 9       cPong(i-1);
10     return i;
11   }
12   native static void cPong(int i);
13 }
(bash) cat -n PingPong.c|tail -n 12
14 #include <jni.h>
15 #include <stdio.h>
16 JNIEXPORT jint JNICALL Java_PingPong_cPong(
17   JNIEnv *env, jclass cls, jint i
18 ) {
19   printf("cPong: %d\n", i);
20   if ( i > 0) {
21     jmethodID mid=(*env)->GetStaticMethodID(env,cls,"jPing","(I)I");
22     (*env)->CallStaticIntMethod(env, cls, mid, i-1);
23   }
24   return i;
25 }
```

The main method at Line 4 calls the jPing method with argument 3, yielding the following stack:

```
main:4 -> jPing(3):7
```

Since i = 3 > 0, control reaches Line 9, where the Java method jPing calls a native method cPong defined in C code as function Java_PingPong_cPong:

```
main:4 -> jPing(3):9 -> cPong(2):19
```

The C function cPong calls back up into Java method jPing by first obtaining its method ID in Line 21, then using the method ID in the call to CallStaticIntMethod in Line 22:

```
main:4 -> jPing(3):9 -> cPong(2):22 -> jPing(1):7
```

After one more call from jPing to cPong, the mixed-language mutual recursion comes to an end because it reaches the base case where i = 0:

```
main:4 -> jPing(3):9 -> cPong(2):22 -> jPing(1):9 -> cPong(0):19
```

This ending condition triggers the following successive returns from CPong(0):19.

```
main:4 -> jPing(3):9 -> cPong(2):22 -> jPing(1):9 -> cPong(0):24
main:4 -> jPing(3):9 -> cPong(2):22 -> jPing(1):10
main:4 -> jPing(3):9 -> cPong(2):24
main:4 -> jPing(3):10
main:5
```

2.1 Compiling with debugging information

Debuggers require that you compile source files with a debug flag. For instance, you can specify `-g` option in `javac` and `gcc` to enable the debugging information when you compile the `PingPong` example.

```
(bash) javac -g PingPong.java
(bash) gcc -g -shared -o libPingPong.so PingPong.c
```

The Jeannie compiler also supports the `-g` debugging option.

```
(bash) jeannie.sh -g JeannieMain.jni
```

If you miss this compile time debug flag, you may not able to inspect data and code at the source-level during the debugging session.

2.2 Launching and terminating a debug session

The Blink debugger master script (`blink.sh`) supports JDB compatible syntax to launch a debugging session. For instance, you can launch the `PingPong` program with the master script as you do with JDB.

```
(bash) jdb PingPong
Initializing jdb ...
> run
...

(bash) blink.sh PingPong
Blink a Java/C mixed language debugger.
(bdb-j) run
...
```

You can terminate the Blink debugging session by running the `exit` command as you do with the JDB.

```
(bash) jdb PingPong
Initializing jdb ...
> exit

(bash) blink.sh PingPong
Blink a Java/C mixed language debugger.
(bdb-j) exit
```

Note that the Blink prompt `(bdb-j)` means that Blink is internally controlling the JDB.

2.3 Setting breakpoint

This section illustrates how to set break points in Java and C code and continue the program execution at each break point. First you begin the debugging session. To set a Java break point, you can use `stop at` command and specify which Java class and which line number within the class file. The Blink `stop at` command is the same as the `stop at` in JDB.

```
(bash) blink.sh PingPong
Blink a Java/C mixed language debugger.
(bdb-j) stop at PingPong:8
```

To set a native break point, you can use the `break` command, which is also available in GDB.

```
(bdb-j) break PingPong.c:20
```

You can query the active break points with `info break` command.

```
(bdb-j) info break
1   java PingPong:8
2  native PingPong.c:20
```

Now you actually run the `PingPong` program, stops the program at each break point and you can continue the execution as follows:

```
(bdb-j) run
jPing: 3
Java break point hit - PingPong.jPing():8
8        if (i > 0 )
(bdb-j) continue
C break point hit - PingPong.c:20
20   if ( i > 0) {
(bdb-c) continue
jPing: 1
Java break point hit - PingPong.jPing():8
8        if (i > 0 )
(bdb-j) continue
C break point hit - PingPong.c:20
20   if ( i > 0) {
(bdb-c) continue
cPong: 2
cPong: 0
```

Note that the Blink prompt `(bdb-c)` means that Blink is internally controlling the GDB.

2.4 Stepping

The `step` command supports source-level stepping for mixed-mode Java/native programs. The Java-only-mode debugger such as JDB ignores stepping into native code as in the following:

```
(bash) jdb PingPong
>stop at PingPong:9
> run
Breakpoint hit: "thread=main", PingPong.jPing(), line=9 bci=29
```

```
9          cPong(i-1);

main[1] where
  [1] PingPong.jPing (PingPong.java:9)
  [2] PingPong.main (PingPong.java:4)
main[1] step
>
Step completed: "thread=main", PingPong.jPing(), line=7 bci=3
7        System.out.println("jPing: " + i );

main[1] where
  [1] PingPong.jPing (PingPong.java:7)
  [2] PingPong.cPong (native method)
  [3] PingPong.jPing (PingPong.java:9)
  [4] PingPong.main (PingPong.java:4)
```

The Blink `step` command takes care of stepping from Java code to native code as follows:

```
(bash) blink.sh PingPong
Blink a Java/C mixed language debugger.
(bdb-j) stop at PingPong:9
(bdb-j) run
jPing: 3
Java break point hit - PingPong.jPing():9
9          cPong(i-1);
(bdb-j) where
  [0] PingPong.jPing  (PingPong.java:9) Java
  [1] PingPong.main  (PingPong.java:4) Java
(bdb-j) step
Java_PingPong_cPong (env=0x805d248,cls=0xbfae04fc,i=2) at PingPong.c:18
18 ) {

(bdb-c) where
  [0] Java_PingPong_cPong  (PingPong.c:18) C
  [1] PingPong.jPing  (PingPong.java:9) Java
  [2] PingPong.main  (PingPong.java:4) Java
```

The `next` command is similar to the `step` command, but it hops one method/function calls within the current line instead of stepping into them. The Blink `next` is also different from the JDB `next` as it takes care of the transition between Java code and native code.

2.5 Examining the stack

Blink supports `where`, `up`, `down` and `locals` commands to examine the call stack. Suppose that you hit a break point at `PingPong.c:20` in the second time.

```
(bash) blink.sh PingPong
Blink a Java/C mixed language debugger.
(bdb-j) break PingPong.c:20
the break point is delayed until the shared library loading
```

```
(bdb-j) run
jPing: 3
C break point hit - PingPong.c:20
20   if ( i > 0) {
(bdb-c) continue
jPing: 1
C break point hit - PingPong.c:20
20   if ( i > 0) {
```

The `where` command discovers the mixed Java/native call frames at the break point.

```
(bdb-c) where
  [0] Java_PingPong_cPong  (PingPong.c:20) C
  [1] PingPong.jPing  (PingPong.java:9) Java
  [2] Java_PingPong_cPong  (PingPong.c:22) C
  [3] PingPong.jPing  (PingPong.java:9) Java
  [4] PingPong.main  (PingPong.java:4) Java
```

The `locals` command allows you to inspect local variables of the currently chosen call frame.

```
(bdb-c) locals
env = (JNIEnv *) 0x805d248
cls = (jclass) 0xbff66de0
i = 0
```

The `up` and `down` commands move the currently selected call frame.

```
(bdb-c) up 1
(bdb-c) locals
i = 1
(bdb-c) up 2
(bdb-c) locals
i = 3
(bdb-c) down 3
(bdb-c) where
  [0] Java_PingPong_cPong  (PingPong.c:20) C
  [1] PingPong.jPing  (PingPong.java:9) Java
  [2] Java_PingPong_cPong  (PingPong.c:22) C
  [3] PingPong.jPing  (PingPong.java:9) Java
  [4] PingPong.main  (PingPong.java:4) Java
```

2.6 Inspecting data with Jeannie expression

The Blink `print <jexpr>` command evaluates a Jeannie expression to examine both Java and native data. Since the Jeannie expression is the superset of Java and C expression, this Jeannie expression evaluator allows the user to inspect both the Java data and native data. Consider the following example.

```
(bash) cd $JAVA_DEV_ROOT/fonda/blink_testsuite
(bash) cat -n CompoundData.java
 1 import java.util.Vector;
 2 public class CompoundData {
 3   static {System.loadLibrary("CompoundData");}
```

```
 4   public static void main(String[] args) {
 5     Vector strings = new Vector();
 6     strings.add("2.50290");
 7     strings.add("2.71828");
 8     strings.add("3.14159");
 9     strings.add("4.66920");
10     double[] doubles = new double[strings.size()];
11     doubles[0] = 3.48;
12     parse(doubles.length, doubles, strings);
13     for( int i = 0; i < doubles.length;i++)
14       System.out.println(doubles[i]);
15   }
16   public static native void parse(int size, double[] doubles, Vector strings);
17 }
(bash)cat -n  CompoundData.c| tail -n22
18 #include <stdlib.h>
19 #include <math.h>
20 JNIEXPORT void JNICALL Java_CompoundData_parse (
21   JNIEnv *env, jclass cls, jint size,
22   jdoubleArray doubles, jobject strings)
23 {
24     int i;
25     /* parse the Java strings to the C doubles. */
26     jdouble* results = malloc( sizeof(jdouble) * size);
27     for(i = 0; i < size;i++) {
28        /* get string. */
29       jclass jvector = (*env)->FindClass(env, "java/util/Vector");
30       jmethodID mid_vget = (*env)->GetMethodID(env, jvector,
                                      "get", "(I)Ljava/lang/Object;");
31       jstring jstr = (jstring)(*env)->CallObjectMethod(env,
                                                 strings, mid_vget, i);
32       /* keep the result. */
33       const char* cstr = (*env)->GetStringUTFChars(env, jstr, 0);
34       results[i] = atof(cstr);
35       (*env)->ReleaseStringUTFChars(env, jstr, cstr);
36     }
37     (*env)->SetDoubleArrayRegion(env, doubles, 0, size, results);
38     free(results);
39 }
```

At the line `CompoundData.c:26`, you can inspect both C and Java data through the Jeannie expression.

```
(bash) blink.sh CompoundData
Blink a Java/C mixed language debugger.
(bdb-j) break CompoundData.c:26
the break point is delayed until the shared library loading
(bdb-j) run
...
C break point hit - CompoundData.c:26
17   if ( i > 0) {
(bdb-c) print size
===> 4
(bdb-c) print ‘(‘strings).size())
===> 4
(bdb-c) print ‘(‘strings).elementCount)
```

```
===> 4
(bdb-c) print bda_cstr(‘(((‘strings).get(1)))
===> "2.71828"
(bdb-c) print ‘(((‘doubles).length)
===> 4
(bdb-c) print ‘(((‘doubles)[0])
===> 3.48
```

Blink also supports debugging Jeannie programs. Consider the following Jeannie example:

```
(bash) cd $JAVA_DEV_ROOT/fonda/blink_testsuite
(bash) cat -n JeannieMain.jni
1 ‘.C {
2 #include <stdio.h>
3 }
4 public class JeannieMain {
5   public static void main(String[] args) {
6     f(1);
7   }
8   public static native void f(int x)
9   ‘{
10     int y = 0;
11     ‘{
12       int z;
13       z = 1 + ‘(y = 1 + ‘(x = 1));
14       System.out.println(x);
15       System.out.println(z);
16     }
17     printf("%d\n", y);
18   }
19 }
```

You can direct the Blink to pause at `JeannieMain.jni:14`:

```
(bash) blink.sh JeannieMain
Blink a Java/C mixed language debugger.
(bdb-j) break JeannieMain$JavaEnvFor_f:14
(bdb-j) run
Java break point hit - JeannieMain$JavaEnvFor_f.c2j1():14
14          System.out.println(x);
```

At the break point, you can inspect the calling context and data.

```
(bdb-j) where
[0] JeannieMain.f(I)V  (JeannieMain.jni:14) Jeannie
[1] JeannieMain.main  (JeannieMain.jni:6) Java
(bdb-j) list
10        int y = 0;
11        ‘{
12          int z;
13          z = 1 + ‘(y = 1 + ‘(x = 1));
14 =>       System.out.println(x);
15          System.out.println(z);
16        }
17        printf("%d\n", y);
```

```
18      }
19    }
(bdb-j) print x
===> 1
(bdb-j) print `y
===> 2
(bdb-j) print z
===> 3
(bdb-j) print x + `y  + z
===> 6
(bdb-j) print x = `y  + z
===> 5
```

3 Tools

This section describes the command line tools for debugging JNI and Jeannie programs. In the normal case, you should only need to use one of them: the "master script" `blink.sh`.

3.1 blink.sh

NAME `blink.sh` – Blink debugger master script.

SYNOPSIS `blink.sh` [*options*] *CLASS* [*arguments*]

PARAMETERS

options Options may be in any order. See OPTIONS below.

CLASS The main class name to launch the debugging session.

arguments The arguments to the main class.

DESCRIPTION

The `blink.sh` master script supports JDB-like command line syntax. This master script first ensures that the necessary environment variables are properly set and that the necessary Blink related files are properly installed. It also ensures that the Blink debug agent code exists in the system and launches the Blink debugger to begin a debugging session.

OPTIONS

`-help` Print out the following message and exit.

`-sourcepath` <*directories separated by* `":"`>
Directories in which to look for source files.

`-v | -verbose[:class|gc|jni]`
Turn on verbose mode in the target debugee JVM.

`-D<name>=<value>`
Set a system property in the target debugee JVM.

`-classpath` <*directories separated by* `":"`>
Set a class path in the target debugee JVM.

`-X`<*option*>
Non-standard VM option in the target debugee JVM.

ENVIRONMENT

`JAVA_DEV_ROOT`
The xtc installation path.

`CLASSPATH`
Paths where to search for the user class files. See the `-classpath` command line option above for details.

`JAVA_HOME`
Path where the JDK is installed.

3.2 Blink Debugger

NAME `xtc.lang.blink.Blink` – Launching a Blink debugging session.

SYNOPSIS `java xtc.lang.blink.Blink` [*options*] *CLASS options*

PARAMETERS

options Options may be in any order. See OPTIONS below.

CLASS The main class name to launch the debugging session.

arguments The arguments to the main class.

DESCRIPTION

The Blink debugger launches the debuggee JVM with its main class name arguments, and starts the mixed-mode debugging session. This debugger usually gets invoked from the blink.sh master script, but you can run this Blink debugger directly by providing the class and library paths for the Blink debugging agent. You can specify these class and library paths with `-bacp` and `-balp` options.

OPTIONS

`-bacp` *path*
: Use the specified directory as home directory for the Java byte code of the Blink debug agent.

`-balp` *path*
: Use the specified directory as home directory for the native code of the Blink debug agent.

`-help`
: Print out this message and exit.

`-sourcepath <`*directories separated by* `":">`
: Directories in which to look for source files.

`-v | -verbose[:class|gc|jni]`
: Turn on verbose mode in the target debugee JVM.

`-D<name>=<value>`
: Set a system property in the target debugee JVM.

`-classpath <`*directories separated by* `":">`
: Set a class path in the target debugee JVM.

`-X<`*option*`>`
: Non-standard VM option in the target debugee JVM.

ENVIRONMENT

`JAVA_DEV_ROOT`
: The xtc installation path.

`CLASSPATH`
: Paths where to search for the user class files. See the `-classpath` command line option above for details.

`JAVA_HOME`
: Path where the JDK is installed.

3.3 Blink command syntax

Within the Blink debugging session, you can control the debugee with the following Blink commands.

Help

help Print help message.

Controlling execution

`run` Start the program execution.

`continue` Continue running.

`step` Execute until another line reached.

`next` Execute the next line, including function calls, and stop.

`exit` Exit the Blink debugger.

Pausing the execution

`break [file:line]`
Add a break point. For instance, `break Main.jni:9`.

`stop at <classid>:<line>`
Add a break point. For instance, `stop at Main:15`.

`stop in <classid>:<method>`
Add a break point. For instance, `stop in Main:main`.

`info break`
List break points.

`delete [n]`
Delete a break/watch point with id `[n]`.

Inspecting the calling context

`where` Dump stack trace

`up [n]` Select n frames up

`down [n]` Select n frames down

`list` Print source code.

`locals` Print local variables in selected frame

Inspecting data.

`print <jexpr>`

Evaluate a Jeannie expression, and print the result. Note that Jeannie expressions include both Java and C expressions.

Index