Charles University in Prague
Faculty of Mathematics and Physics
Department of Distributed and Dependable Systems

Tomáš Martinec

User manual for the MSIM debugger

(this manual is derived from the thesis)

Last update: November 10, 2013

1. User manual

Users are supposed to understand debugging concepts in general. This manual helps with the initial setup and describes more important GUI elements of the Eclipse IDE.

1.1 Downloads

The user needs to get these packages:

1. Eclipse with appropriate plugins

   Eclipse Classic (or For Java Developers) 4.3.1 is required. Newer versions of eclipse are likely to work too, but it is not guaranteed. Note that you should not use official CDT plugin for C/C++ development, because our own version will be installed. The packages will be installed through the Eclipse installation dialog.

   The Eclipse packages are also available on our servers:

   32-bit versions:
   `http://aiya.ms.mff.cuni.cz/~martinec/eclipse/eclipse-standard-kepler-SR1-linux-gtk.tar.gz`
   `http://aiya.ms.mff.cuni.cz/~martinec/eclipse/eclipse-standard-kepler-SR1-win32.zip`

   64-bit versions:
   `http://aiya.ms.mff.cuni.cz/~martinec/eclipse/eclipse-standard-kepler-SR1-linux-gtk-x86_64.tar.gz`
   `http://aiya.ms.mff.cuni.cz/~martinec/eclipse/eclipse-standard-kepler-SR1-win32-x86_64.zip`

2. MSIM that supports GDB debugging

   Unfortunately, the distribution packages are not available now. Users have to build MSIM from sources under the *bazaar* version system. The sources can be obtained from Launchpad and use the following command to do so:

   ```
   bzr branch lp:~fyzmat/msim-private-tm/trunk
   ```

3. patched GDB-7.6.

   Currently, a special version of GDB must be built from sources too. Here is the source package:

   `http://aiya.ms.mff.cuni.cz/~martinec/gdb/gdb-7.6-patched-msim-debugger.tar`

1.2 Installation

1. Unpack the Eclipse Classic, run it and choose your workspace folder.

2. Open the menu *Help -> Install new software* and add the following update site:

   `http://aiya.ms.mff.cuni.cz/~martinec/msim-debugger/msim.debugger.update.site`.

   Uncheck *Group items by category*, select all the plugins and run the installation.

3. Build MSIM by following commands:

   ```
   cd /path/to/msim/sources
   ./configure
   make
   ```

   After these commands you should see built binary of MSIM in the *bin* directory. Put the built binary into the same directory where your *msim.conf* is located.

   For cygwin users: A problem with linking the *readline* library was encountered in *Cygwin*.

   The *readline* is located in the *ncurses* library. If the *ncurses* was missing the *configure* script might not recognize the *readline* library. As a quick fix change the following line in the *configure* script:

   ```
       #original line:
   LIBS="-lreadline $LIBS"

       #changed line:
   LIBS="-lreadline -lncurses $LIBS"
   ```

   Additionally, add the *-lncurses* option to the *LIBS* variable in `src/Makefile`. Also, you might have to change the following lines in that makefile:

   ```
       #original lines:
       $(TARGET): $(OBJECTS) $(DEPEND)
         $(CC) $(CFLAGS) $(LIBS) -o $@ $(OBJECTS)

       #changed lines:
       $(TARGET): $(OBJECTS) $(DEPEND)
         $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(LIBS)
   ```

4. Build GDB by following commands:

   ```
   cd /path/to/gdb/sources
   ./configure --target=mips
   make
   ```

   These commands should create the GDB executable *gdb* in the *gdb* directory.

1.3 Setting up a new project

Copy sources of your program for MSIM to the chosen workspace. Open the Eclipse and create a new project. Choose *Makefile project with existing code* under the *C/C++* group. Select the toolchain that is appropriate for your platform (*Linux GCC* for linux and *Cygwin GCC* for windows).

Now you should see the structure of your project in *Package Explorer* or in *Project Explorer*. Change the optimization flags in your makefiles to *-O0* and add the *-g* option for GCC. Rebuild your project. The project should be built according to the rules of your makefile.

Open *Run -> Debug Configurations...* dialog. Create a new *C/C++ MSIM Application* launch configuration. Fill in the name of your project and the debugged binary that contains debugging information. For example, *kernel.elf* or *kernel.raw*. Note that the default Eclipse binary search will miss **.raw* files, because it is not very usual suffix. Thus, the binary might have to be specified manually.

Switch to the *Msim launch options* tab. Here you have to configure paths for the built MSIM and GDB. The GDB ini file is not usually needed. Write the name of your main function, if you want to stop in it after the launch.

Debugging should work now. You can launch the debugging session.

For cygwin users: The source filenames are referenced from the root (/) directory and the Eclipse might not be able to locate them. Thus, you will have to specify their location after the first launch. Click on the *Edit source lookup path* in the code editor after launching and add *Filesystem directory* to your cygwin installation (e.g. *C:\cygwin*).

1.4 Debugging views

This section contains screenshots of important GUI views. The relevant GUI elements are marked and described.

Views can be activated in the *Window -> Show view* menu.

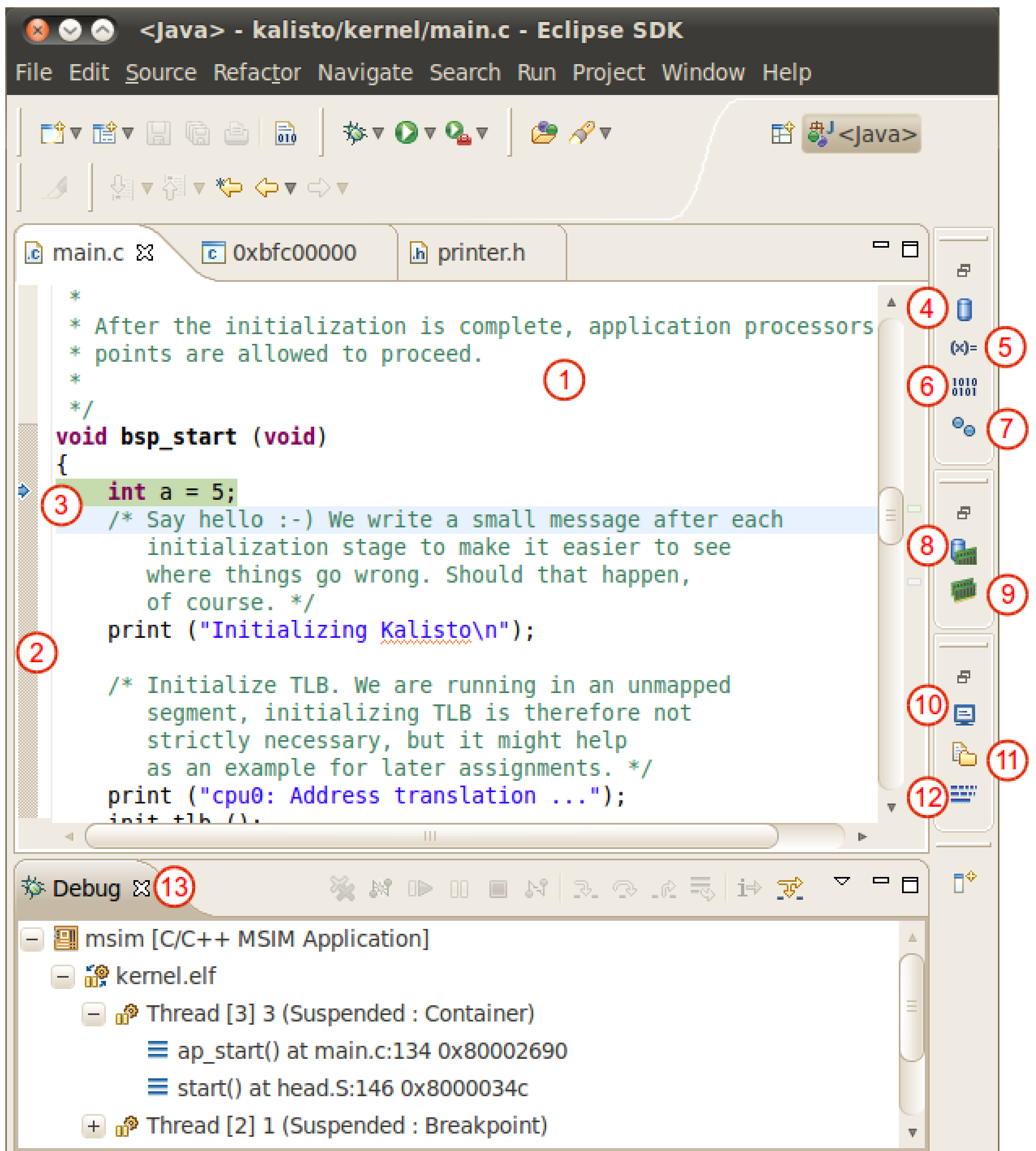

Figure 1.1: **Source view with other views in the side toolbar**.

	Name	Notes
1	Source view	
2	Place for breakpoints	Right click to open menu and toggle or enable/disable source-level breakpoints
3	Line where the debugged thread is stopped	
4	Memory Browser view	
5	Variables view	
6	Registers view	
7	Breakpoints view	
8	TLB Contents view	
9	Physical Memory view	
10	Console view	
11	Project Explorer view	
12	Disassembly view	
13	Debug view	

Table 1.1: **Marked GUI elements of the view 1.1**.

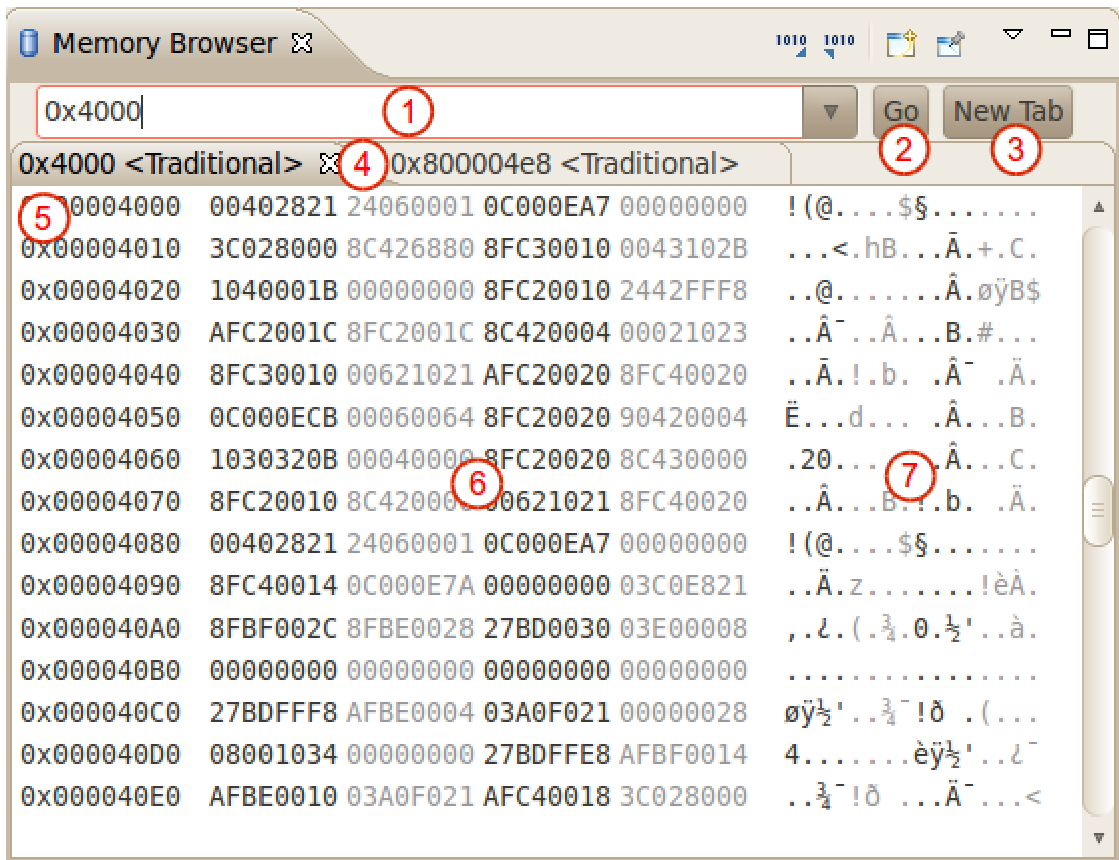

Figure 1.2: **Memory browser view**.

	Name	Notes
1	Expression input	You can type here any expression that specifies an address
2	Go to the specified address	
3	Create a new tab	
4	Tabs for browsing memory	
5	Address column of the tab	
6	Column with hex-dumped memory	It is possible to change the memory by writing desired hexadecimal values in this column.
7	Column with ASCII-dumped memory	It is possible to change the memory by writing desired ASCII chars in this column.

Table 1.2: **Marked GUI elements of the view 1.2**.

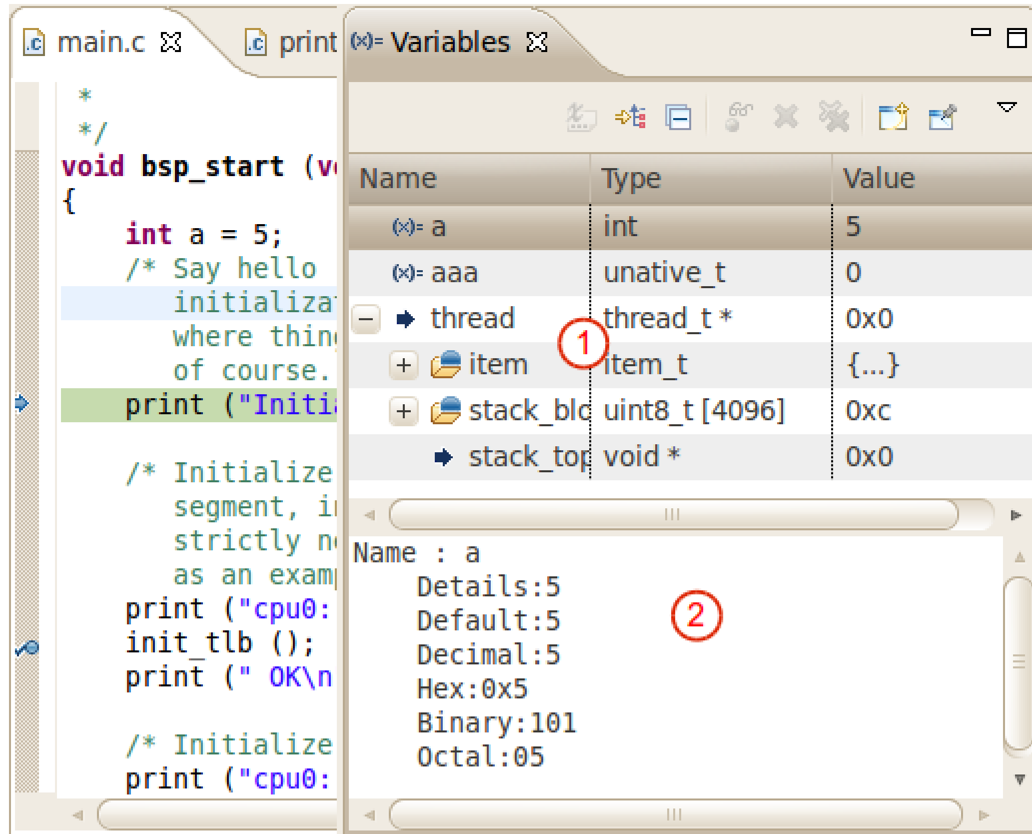

Figure 1.3: **Variables view**.

	Name	Notes
1	Table with local variables	The rows contain an identifier, a type, and a value of the related local variable. The user can change values of variables in the last column.
2	Details for values	

Table 1.3: **Marked GUI elements of the view 1.3**.

	Name	Notes
1	Table with registers	The rows contain a name of the register and its value. The user can change values of registers in the column with values. Changed registers are colored.
2	Details for values	

Table 1.4: **Marked GUI elements of the view 1.4**.

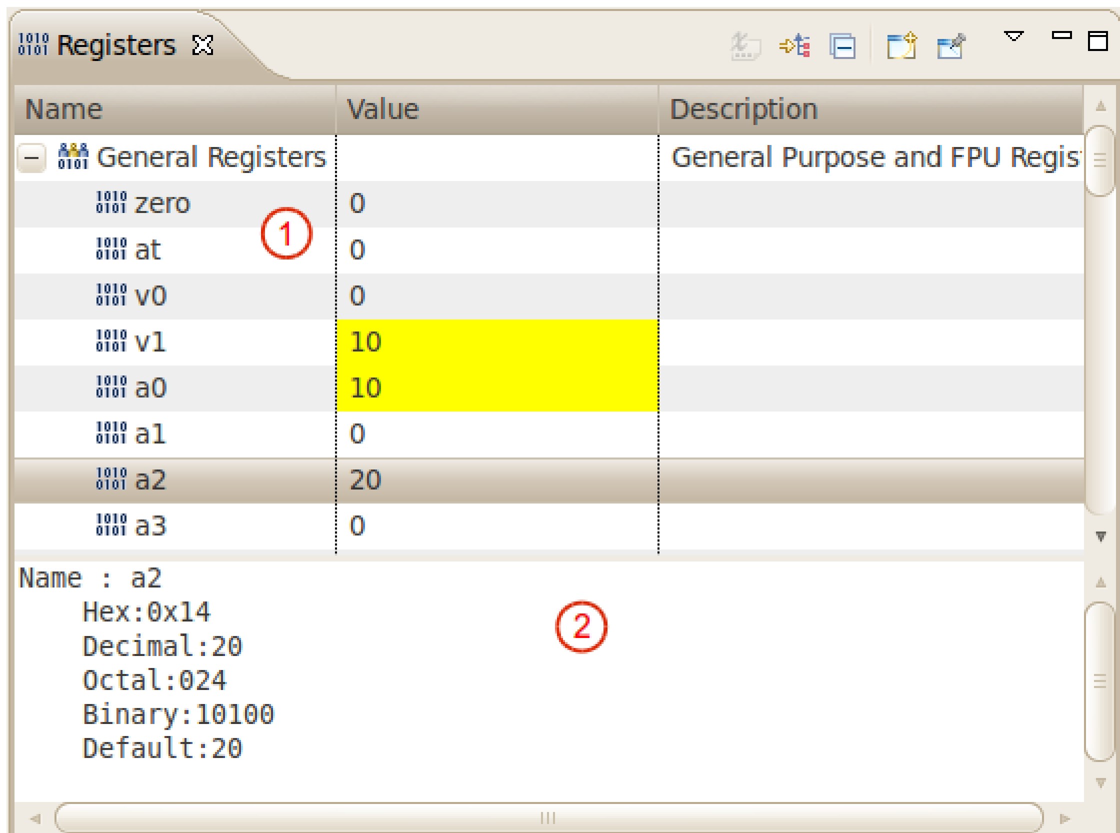

Figure 1.4: **Registers view**.

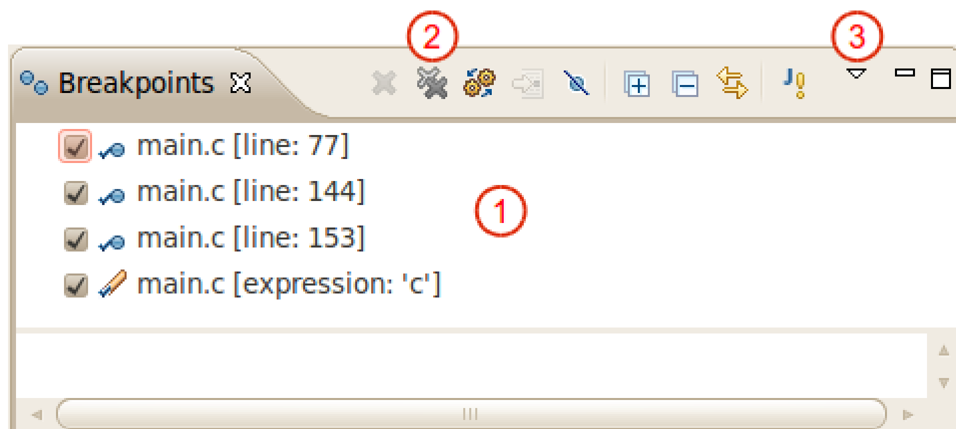

Figure 1.5: **Breakpoints view**.

	Name	Notes
1	List of breakpoints	Both normal and memory breakpoints are listed.
2	Remove all breakpoints	
3	Additional options	The user can set a memory breakpoint in this menu. Another way of setting a memory breakpoint is via the menu *Run -> Toggle Watchpoint*.

Table 1.5: **Marked GUI elements of the view 1.5**.

TLB Contents

Index	Page	Mask	G	ASID	V	D	Frame	C
0a	0x0	0xffffe000	0	0xff	0	0	0x0	0x0
0b	0x1	0xffffe000	0	0xff	0	0	0x0	0x0
1a	0x0	0xffffe000	0	0xff	0	0	0x1000	0x0
1b	0x1	0xffffe000	0	0xff	0	0	0x2000	0x0
2a	0x0	0xffffe000	0	0xff	0	0	0x2000	0x0
2b	0x1	0xffffe000	0	0xff	0	0	0x4000	0x0
3a	0x0	0xffffe000	0	0xff	0	0	0x3000	0x0
3b	0x1	0xffffe000	0	0xff	0	0	0x6000	0x0
4a	0x0	0xffffe000	0	0xff	0	0	0x4000	0x0
4b	0x1	0xffffe000	0	0xff	0	0	0x8000	0x0

Figure 1.6: **TLB Contents view**.

	Name	Notes
1	Index column	Each row reflects translation of one page to one frame. Note that a TLB entry for the R4000 processor maps a pair of the following pages to two frames. The used way of displaying a TLB entry is separating it into two rows.
2	Page index	Address of a virtual page divided by size of one page.
3	Mask	TLB hit occurs when *virtual_address* & *mask* == *page_index*. This value is derived from the *PageMask* register. Bits 0-12 are always zeroes, bits 13-24 are set by the debuggee, and bits 25-31 are always ones. See [?] [p. 81] for more details.

4	Global bit	ASID is ignored if this bit is set. Note that this bit is never directly accessed by the debuggee. It is computed during a TLB write as logical AND of *EntryLo0* and *EntryLo1* global bits.
5	ASID	Address space identifier of the entry. Note that the current ASID is stored in the *EntryHi* register.
6	Valid bit	This bit is set if the page-to-frame translation of this row is enabled.
7	Dirty bit	This bit is set if the page is writeable.
8	Frame	Address of the translated physical frame.
9	Coherency bits	Three coherency bits. Not used in MSIM.

Table 1.6: **Marked GUI elements of the view 1.6**.

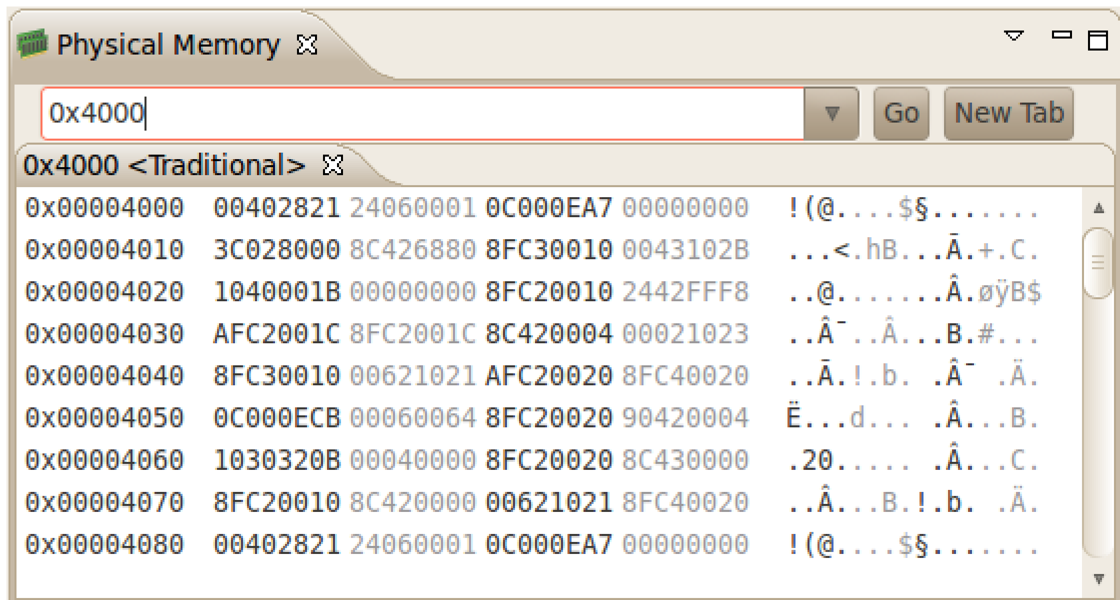

Figure 1.7: **Physical memory view**. The usage is the same as for the *Memory browser view*

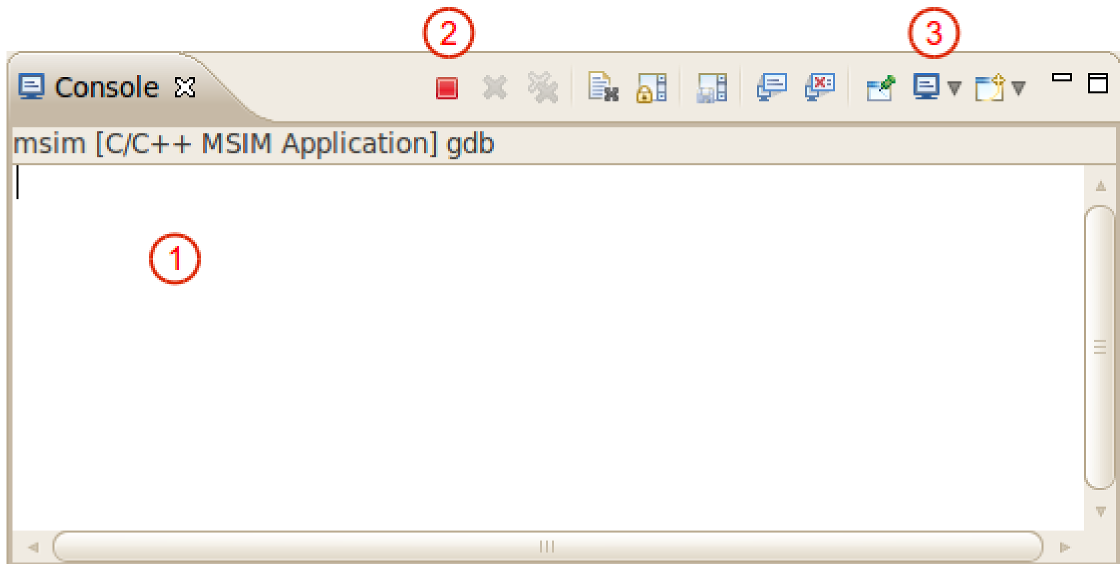

Figure 1.8: **Console view**.

	Name	Notes
1	Console output	Useful for seeing MSIM output or GDB/MI communication.
2	Terminate debugging session	
3	Select another console	

Table 1.7: **Marked GUI elements of the view 1.8**.

Figure 1.9: **Project explorer view**. The Kalisto project is currently loaded.

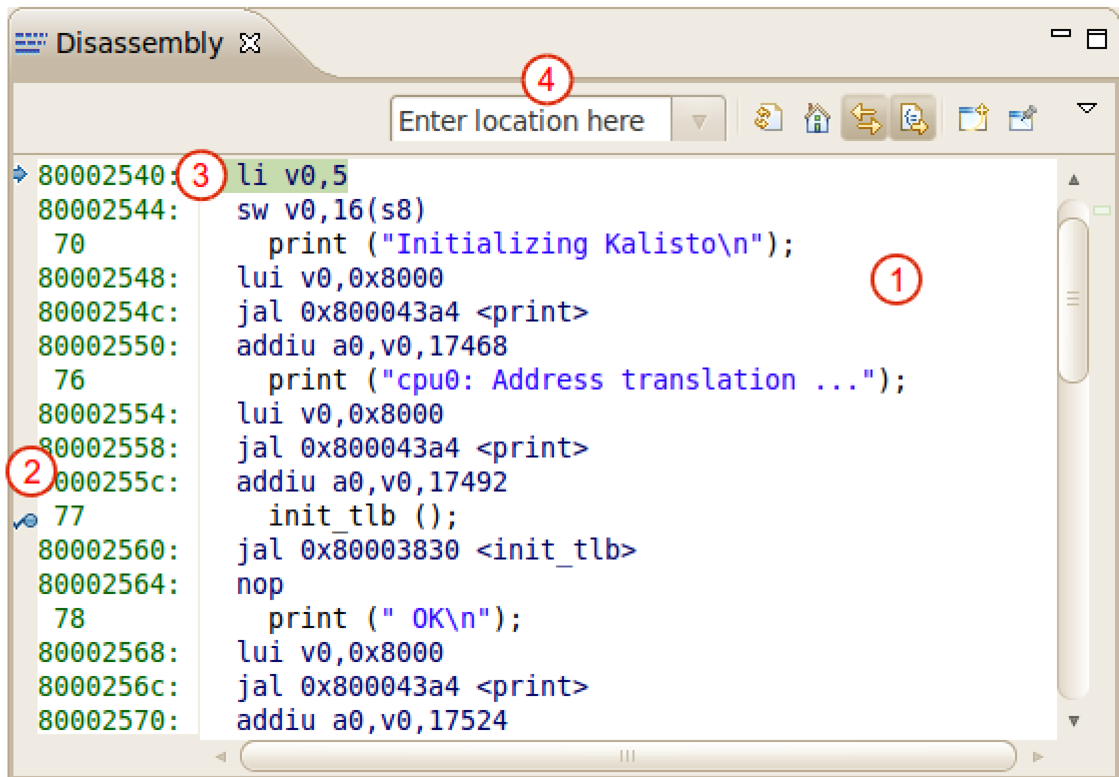

Figure 1.10: **Disassembly view**.

Name		Notes
1	Disassembly view	C source is merged into the instructions. The first column holds addresses of instructions or lines of the C code. The second column contains instructions and the C code. Symbols are added to the known addresses.
2	Place for breakpoints	Right click to open menu and toggle or enable/disable instruction-level breakpoints.
3	Line where the debugged thread is stopped	
4	Search input	Any expression that specifies an address can be typed here.

Table 1.8: **Marked GUI elements of the view 1.10**.

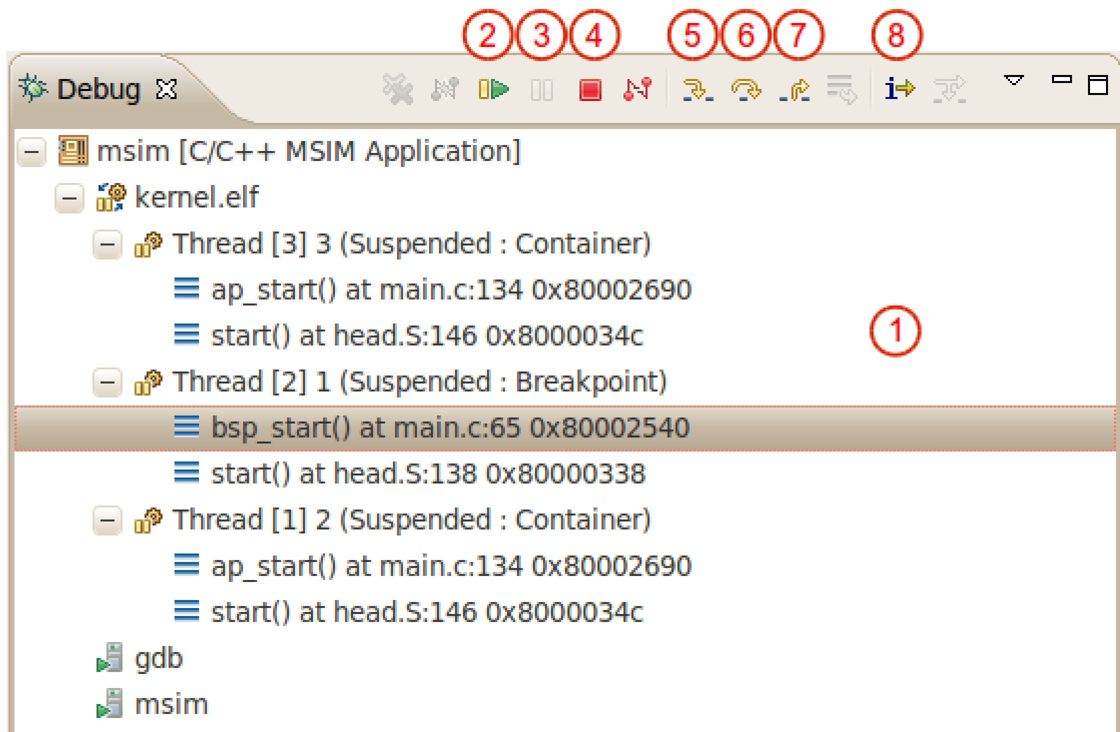

Figure 1.11: **Debug view**.

	Name	Notes
1	Debug view	All the processes, their threads and call stacks are listed. The user selects the current thread by choosing it in this view.
2	Resume	
3	Interrupt	
4	Terminate debugging session	
5	Step into	
6	Step over	
7	Step out	
7	Toggle instruction-level debugging	This enables the user to do instruction-level stepping.

Table 1.9: **Marked GUI elements of the view 1.11**.