NeuroMatrix(r) SDK

Multitarget JIT Debugger.

User's Guide.

30047-01 95 02 A

RC Module [http://www.module.ru]

NeuroMatrix(r) SDK: Multitarget JIT Debugger.

by RC Module [http://www.module.ru]

Table of Contents

Preface

This document describes Multitarget JIT Debugger (emudbg) 1.2, english version.

Chapter 1. Introduction

Multitarget JIT Debugger is a part of the NeuroMatrix® NM6403 SDK by RC Module.

The Debugger is a windowing tool that enables NM6403 program debugging on the local target from the Win9x/WinNT host machine.

1. Changeable Debugging Targets

The Debugger uses changeable debugging targets. A debugging target communicates with the host debugger through some type of universal processor interface to enable cross debugging. A debugging target covers all debugging features of the target processor. Change debugging target to change the debugging environment.

For example, NeuroMatrix® NM6403 SDK 1.20 includes the following debugging targets: functional (instruction level) NM6403 emulator and NM6403 processor on PCI (NM1) board.

In addition to universal processor interface, a debugging target gives to the debugger some type of specific information. For example, using NM6403 emulator you can control VU registers and memory unaccessible on the hardware processor.

The Debugger requires at least one target, because the Debugger has no included debugging targets.

2. Debugging an already-running program

The Debugger can be attached to a process on the connected target that is already running (JIT (Just In Time) debugging).

Also a debugging target has to support JIT debugging.

Just In Time debugging includes the following steps:

1. The Debugger detects an already running process on the connected debugging target,
2. The Debugger stops process, connects to debugging target and attaches to the process,
3. This step includes a general debugging session: the user controls debugging process state and starts any debug actions.
4. A debugging session is finished: the debugging process is detached and is run, the debugger disconnects from the debugging target.

For execution of the last step, the Debugger has a specific debugging action: run and disconnect.

3. Debugging features

The Debugger supports the basic features and commands for a user program debugging.

The Debugger provides the following services:

- controls program execution, using Instructions or Source windows,
- Displays and modifies register value and memory contents,
- Displays call stack and virtually unwinds the call stack,
- Displays and modifies global objects of the program (function addresses and global variables),
- Displays and modifies local variables of the frames,
- Uses target specifics information which is provided by debugging target,
- Sets conditional/unconditional breakpoints.

For more information see debug actions, Debugger windows, breakpoints

4. Executable file format

The Debugger uses ELF (Executable and Linking Format) format files and DWARF (Debug With Arbitrary Record Format) 2.0 debug information. DWARF 1.0 debug information is not supported.

5. Current version limitations

The current version of the debugger (see version number in contents [vii]) has the following limitations:

- An internal C++ expression evaluator used for variable value modification, accepts C++ variable names only.
- For conditional breakpoints you can input any valid C++ language expression, but the debugger does not use the evaluation results (because of the evaluator imperfection).
- Breakpoints are not restored after program restart.
- The Globals window displays the current compilation unit objects only.
- The debugger information protocol cannot be saved in the file.

These limitations will be addressed in the next versions of the debugger.

Chapter 2. Actions

The debugging commands are displayed in the Debug menu. They include different steps, set/clear breakpoints, run, stop, restart, and also a special command - run and disconnect.

All commands (except program stop) can be executed when the debugging program is paused.

Debugging commands are listed here with their descriptions, toolbar buttons, menus and shortcut keys.

1. Single Stepping: Step into and Step over.

A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

The debugger can make steps through code in two basic ways:

- by machine instructions;
- by source lines.

Step by source line lets you step through statements in your source code. Step by source line requires debug information and source files.

The next statement for step by source line depends only on debug information. Debug information statement order can mismatch with the real statement order.

The debugger automatically steps through your code at the instruction level if the disassembly window has focus when you choose the step command. If the source window has focus, the debugger steps over the code line. If some other window has focus the step mode is not changed.

The current step mode is displayed in the status bar.

1.1. Step into

Command: | Debug/Step into | **F11**.

The Step into command makes the debugger walk through your code one statement or instruction at a time.

Behavior of stepping into by source lines has the following features. If the debugger encounters a function call, it steps into the called function if that function has debugging information (about source lines). When function execution is finished, single-step execution returns to the caller. If the function has no debugging information (i.e. source line information is not available), the debugger executes the function by default until the next source line is encoutered (wherever it will be).

1.2. Step over

Command: | Debug/Step over | **F10**.

The Step over command executes the program statement highlighted by the execution point and advances the execution point to the next statement. The execution point indicates the next executable line in your source code or machine instruction. The debugger steps over function calls while executing them as a single unit and stops program at return points. If you issue the Step over command when the execution point is located on a function call, the debugger runs that function at full speed, then positions the execution point on the statement that follows the function call.

2. Step out

This command is available only if there is debugging information at the execution point.

Command: {↑} | Debug/Step out | **Shift+F11**.

The Step out command executes the current function and halts when execution returns to the function's caller.

3. Run to cursor

The current window must be a disassembly or a source window so that the location to execute can be determined.

Command: →I | Debug/Run to cursor | **Ctrl+F10**.

When you run to the cursor, your program is executed at full speed, then pauses and places the execution point on the code line containing the cursor.

4. Run

Command: ⚡ | Debug/Run | **F5**.

The Run command executes your program continuously until either a breakpoint is encountered or the program is interrupted by the user or the program terminates.

5. Animate

Command: 🎥 | Debug/Animate | **F8**.

The Animate command is a self-repeating Step into command. Instructions or source lines are executed continuously until either a breakpoint is encountered or the program is interrupted by the user or the program terminates. The Debugger's display changes to reflect the current program state between each trace (animation), which allows you to watch the flow of control in your program.

6. Restart

Command: 🔄 | Debug/Restart | **Ctrl+F5**.

This command resets a program to its original program entry point.

This command, like the load command, reloads code and data sections from the local executable file. if the current program has no association with the local executable file, it can not be restarted.

Note

The state of the vector unit of the processor is not cleared by restart. This can lead to errors if the program assumes clear initial state of VU. If that is the case one could use Reset command (which perform real processor reset) in place of restart.

7. Run and disconnect

Command: Debug/Run and disconnect .

By this command, the debugging program on the current target runs and the debugger breaks connection with the current target.

Run and disconnect command is very important, because it finishes the JIT debugging cycle. Debugging target disconnection is a complex operation which can be executed only when the debugger totally controls the target, i.e. when the user's program is stopped at the target. That is why the last step of the cycle is implemented using one composite operation.

8. Stop

This command is only available when the debugging program is run.

Command: | Debug/Stop | **Ctrl+Break**.

The Stop command interrupts the debugging program execution.

The Stop command is a complex operation during which the debugging target with its program should be in the operational state. If the debugger has no target response during timeout (5 seconds by default) wait, the Stop command is considered incompleted. In this case the debugger state is not changed: the program is run.

A failed Stop command does not necessarily mean an error in the target state and the program. For example, the target for remote debugging through, say, TCP/IP protocol, can just be disconnected for a while from the monitor on the remote computer.

9. Toggle breakpoint

The current window must be a disassembly or a source window.

Command: | Debug/Toggle breakpoint | **F9**.

This command adds or removes a breakpoint at an instruction or a line at the cursor position in the current window. If a breakpoint exists on the selected statement, then Toggle breakpoint will delete the breakpoint at that code location.

If there is no debug information about addresses for the current source line, the breakpoint is set on the next source line which has this debug information.

When you choose Toggle breakpoint, the debugger sets an unconditional (simple), breakpoint at the instruction that you have selected in the current window. A simple breakpoint has no conditions, and the only action is that it will pause the program's execution. The conditions can be added trough Breakpoint Dialog (View/Breakpoints).

Chapter 3. Debugging targets, debugging target libraries

Debugging target is a module implementing universal processor interface for the target processor. Debugging targets are in the external modules - debugging target libraries. Debugging target library is a dll. When a target from a library is needed, the library is connected by the debugger. A library can consist of one or more targets.

Using different targets you can debug programs running on different target devices or program emulators.

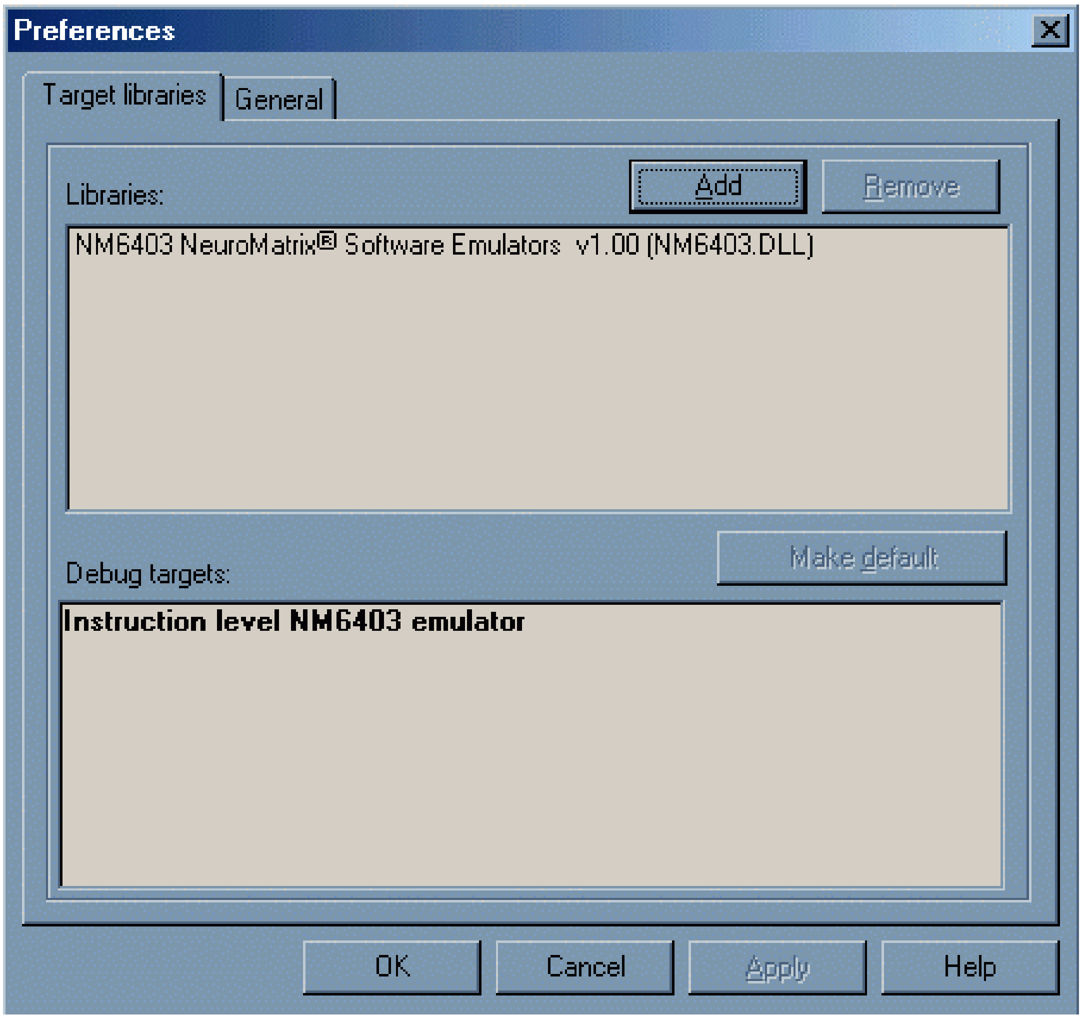
Thus program debugging can take place step by step - each time in the most convenient environment. For example, at first it can be on an emulator (or emulators) - in the slowest environment allowing, however, to work with information unavailable by debugging on a physical device. Then it can perhaps proceed on a debugging variant of the device that has more capacity of processor state monitoring and program control than a serial device. At last it can proceed in the similar to real conditions on a serial device.

1. Target library registration

Before using a debug target, the debugger must register a debug target library with this target.

Command:Edit/Preferences..., page "Target libraries"

Page "Target libraries" display the list of registered target libraries and the list of debug targets in these libraries. Also this page allows to add or remove a debug library and make it default. The debugger connects a default target automatically at the start.



Libraries:

NM6403 NeuroMatrix® Software Emulators v1.00 (NM6403.DLL)

Debug targets:

Instruction level NM6403 emulator

A default target is marked by bold font.

Add

To add a new target library to the list, choose this command. This command displays Open dialog box. You can open multiple libraries (dll files). If the opened dll is a target library and the registered library target list does not have it, this dll will be added to the list.

Remove

The chosen target library will be removed from the list and all it's targets will become unavailable. You can't remove a library containing a current connected target.

Make default/No default

This command sets or unsets a default target.

New targets are available for connection and further use after registration.

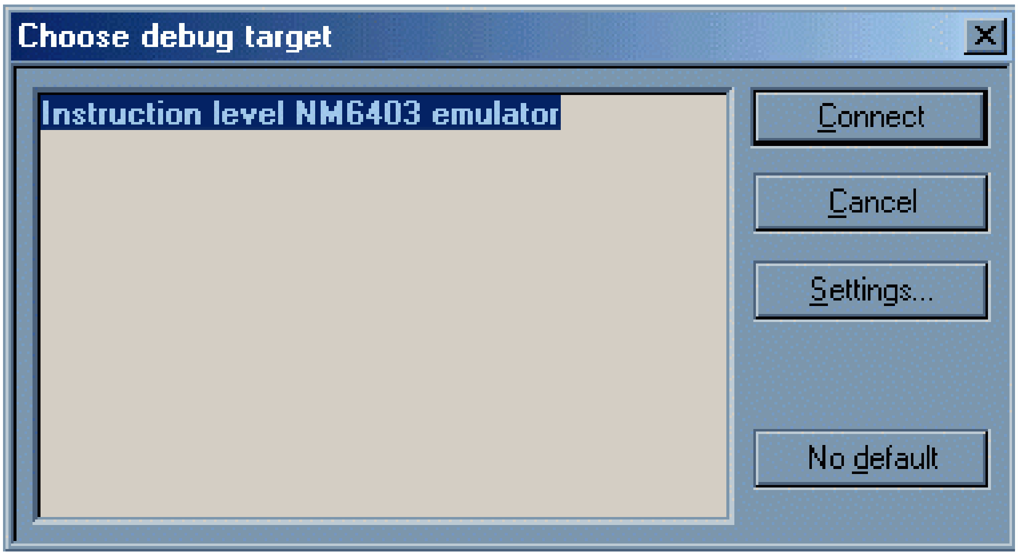
2. Connect to debug target

You can connect to a new debug target at any time during a debug session.

If the debugger has already connected to a debug target and this target has an active process, the current debug session will be finished. You can leave the active process on the old target. In this case the debugger stops the active process and executes Run and disconnect command.

The debugger handles all abnormal situations during target connecting/disconnecting. Cause messages are produced to the debugger protocol.

Command: Target/Connect to...



A default target is marked by bold font.

Connect

The debugger connects to the chosen debug target.

Setting...

This command displays start settings dialog for the chosen target. The Settings values will be kept in the user's area of the system registry (HKCU) and will be used for target initialization, when the debugger connects to the debug target.

The dialog view and the setting are defined by the debug target.

Make default/No default

This command sets or unsets a default target.

2.1. Additional information:

The debugger keeps the target libraries list in the system registry under the following key: HKLM/<debugger_key>/targetLibraries/.

Target initialization settings are stored under the following key: HKCU/<debugger_key>/targetPrefs/.

A default target option is stored under the following key: HKCU/<debugger_key>/targetPrefs/defaultTarget.

3. Reset

The target reset command initializes the target processor to power-up state. If the debugger interacts with the target processor with the help of a mediator (for example via the debugging monitor on the target processor), the target reset is also necessary to restore the connecting chain components functioning.

Command: Target/Reset

The current debug target is reset. You can select to load a current program again after target reset.

4. Load program

Use this command to load a program from a valid ELF file onto your actual/simulated debug target. Program segments are located in the memory according to executable file information. A new program replaces the old one, because only one active process is available per debug target.

You will be prompted with the File Open dialog box where you can select the desired file.

Command: |Target/Load program |**Ctrl**+**O**,

Also you can load a program from the recent file list in the Target menu.

5. Unload program

Command: |Target/Unload program |**Ctrl**+**U**,

This command unloads a program from the debug target (the debug target discards the program).

6. Target specifics window

Command: View/Target specifics

The current debug target displays its specific information in this window. For example, NM6403 emulator displays vector registers, vector matrixes and other VU devices of the processor.

The debugger is incapable of placing the reference information about GUI of the target specificity window in its help system. The info about targets is distributed together with libraries containing these targets. This info can be usually found in the same directories as files of libraries.

Chapter 4. Breakpoints

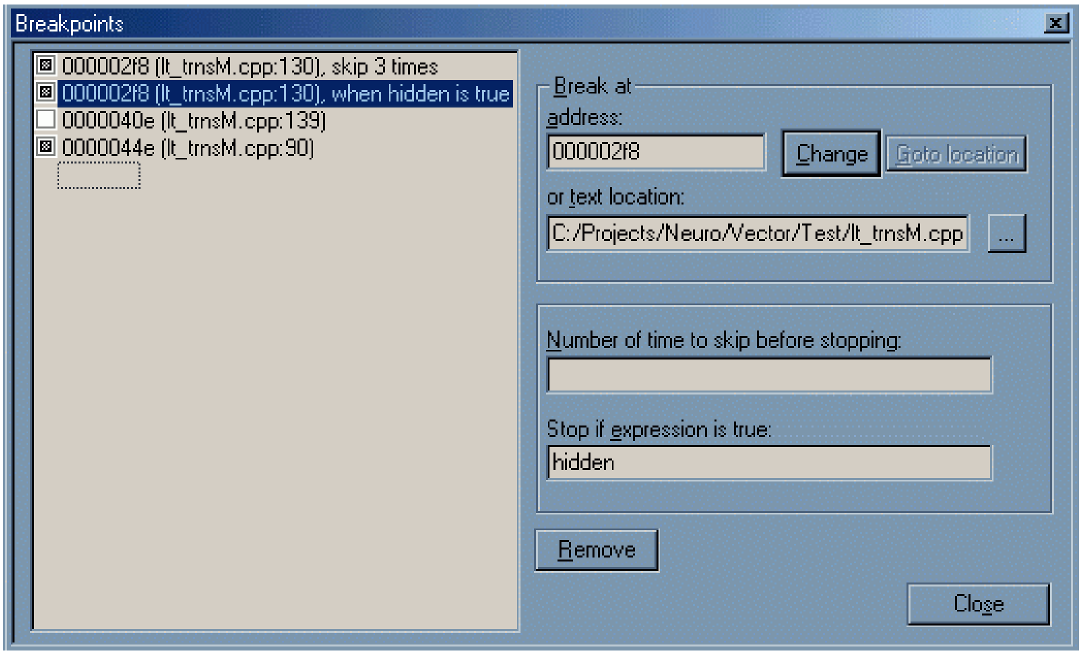
Breakpoints are used to interrupt the work of the debugged user's program to view the current state of the processor memory, registers, call stacks, etc.

You can set a breakpoint to stop your program at any source line or address using Toggle breakpoint command from disassembly and source windows or using the Breakpoints dialog.

Through the Breakpoints dialog breakpoints can also be qualified by setting a pass count that specifies how many times the breakpoint must be passed over before being activated or/and conditions. Condition is a conditional C++ expression that is evaluated each time the breakpoint is encountered. Program execution stops when the expression evaluates to True (non-zero). The accepted syntax of a condition is a limited C++ expression subset (see Current version limitations).

1. Breakpoints Dialog

Command: View/Breakpoints



Breakpoints

- ☒ 000002f8 (lt_trnsM.cpp:130), skip 3 times
- ☒ 000002f8 (lt_trnsM.cpp:130), when hidden is true
- ☐ 0000040e (lt_trnsM.cpp:139)
- ☒ 0000044e (lt_trnsM.cpp:90)

Break at

address: 000002f8

or text location: C:/Projects/Neuro/Vector/Test/lt_trnsM.cpp

Number of time to skip before stopping:

Stop if expression is true: hidden

The complete list of the existing breakpoints and its qualifications are displayed on the left. You can enable/disable a breakpoint by clicking the checkbox on the left on the desired line.

The current edited breakpoint qualifications are displayed on the right. Use the edit controls to change the breakpoint properties. New properties require the following conditions:

- Breakpoint address is a hexadecimal number;
- Text location is a file name and line number separated by a colon. For file name and line number:
  1. file name can be entered without the path,
  2. there are no spaces before and after the colon,
  3. line number is a decimal value,
  4. line order starts from 1;

- The pass count enables you to specify a particular number of times that a breakpoint must be passed for the breakpoint to be activated. Pass count is a decimal number.

Condition expression is validated and evaluated each time the breakpoint is encountered during the program execution. When an expression error occurs, the expression always evaluates to true (or not zero) and the breakpoint pauses the program run.

Change

This command makes breakpoint changes.

Use the last empty line of the breakpoint list to set a new breakpoint. Type the new breakpoint properties in the edit controls on the right of the dialog.

This is a default command. It is executed on **Enter** key press no matter if the command button has focus.

...

This command opens a popup menu with the source file list. Choose one to enter it in the Break at text location edit control.

Remove

Remove the selected breakpoints.

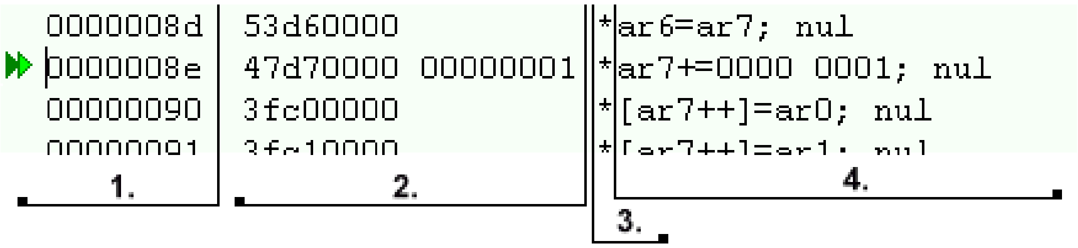
Chapter 5. Debugger windows

Debugger windows display different information about the program state and the debug target state.

Basic debug commands from Debug menu can be executed from any working window.

1. Disassembler window (Instructions)

Command: View/Instructions



```
   0000008d  53d60000           *ar6=ar7; nul
▶▶ 0000008e  47d70000 00000001  *ar7+=0000 0001; nul
   00000090  3fc00000           *[ar7++]=ar0; nul
   00000091  3fc10000           *[ar7++]=ar1; nul
```

1. - instruction address,
2. - instruction code (one or two words),
3. - parallel execution flag (parallel execution bit is set, if '*' symbol *not exist*),
4. - disassembled assembler command.

Left column labels point to:

- the current executive address (▶▶),
- the current frame address (▶…),
- breakpoint on this line (⊤).

In the figure you can see the current executive address label and the text cursor.

If the disassembler window has focus, the step mode is switched to *step by instruction*.

A part of basic debug command appears in window context menu.

1.1. Go to address

Command: **Ctrl**+**G** | <context menu>/Go to... | View/Go to...

Input a desired address (in hexadecimal format) in the appeared dialog box and click the Go to button.

2. Registers window

Command: View/Registers

The Registers window displays the contents of the target processor registers and flags from the state register.

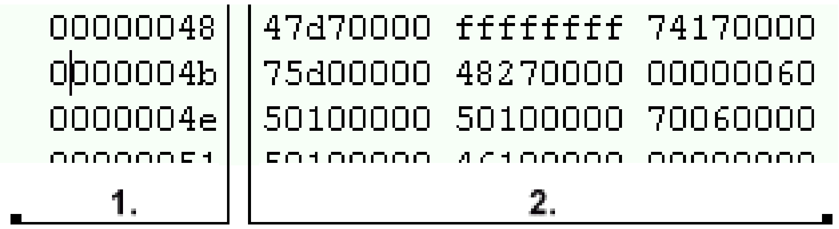
2.1. Modify register value

You can enter a new value for the selected register. New values are sent to the debug target and are copied to the processor registers.

The unchangeable registers do not have editable values.

3. Memory window

Command: View/Memory



```
00000048  47d70000 ffffffff 74170000
0000004b  75d00000 48270000 00000060
0000004e  50100000 50100000 70060000
```

1. - addresses,
2. - data.

The Memory window can display numbers in different formats. For example, NM6403 memory can be displayed in 32-bit or 64-bit word format.

You can execute all Debug menu commands from this window.

3.1. Display format selection

You can control the Memory window display by using options in the context menu. The current format is checked.

3.2. Modify memory values

You can edit memory contents using this window. New memory values are sent to the debug target.

3.3. Go to address

See instructions window.

4. Call stack window

Command: View/Call stack

During a debug session, the Call stack window displays the stack of function calls that are currently active. The calls are listed in the calling order, with the current frame (function) (the most deeply nested) at the top. If the frame has debug information, the window displays the frame name, parameter types and values. If the frame has no debug information, the window displays the frame return address (from the stack).

You can execute all Debug menu commands from this window.

4.1. Virtually call stack unwinding

By double-clicking the left mouse button on the frame line, the Debugger virtually unwinds the call stack to this frame. Here:

- the debugger displays the current line/address of the current watch frame,
- The Locals window displays local variables of the current watch frame.

The current watched frame is marked with ⏩ sign for frame of execution or ▶… sign for others.

5. Sources windows

Command: View/Sources/

These windows display the program source texts.

Like in the Instructions window, labels point to the current executive line, the current frame line and breakpoint lines.

If the source window has a focus, the step mode is *step by source line*.

You can execute all Debug menu commands from these windows.

6. Globals window

Command: View/Globals

The program global objects are:

- functions (including member functions),
- global variables,
- static class members.

The Globals window contains a spreadsheet with fields for the variable name and value.

All global names are marked with:

- curly braces for classes,
- green square for data,
- red square for function.

The complex type variables are marked with a box containing a + sign in the Name column. You can expand the variable by clicking the sign box, which opens into a tree that may contain additional boxes. When a variable is expanded, the box in the Name column contains a − sign. You can collapse an expanded variable by clicking the sign box.

6.1. Modifying the value of a variable

To modify the value of a variable in the Globals window select the line containing the variable whose type you want to modify. If the variable is an array or an object, use the sign box to expand the view until you see the value you want to modify. Click the value, type a new value or a C++ expression (see Current version limitations), and press **Enter** or select another spreadsheet line to complete or press **Esc** to cancel.

If the typed expression is incorrect, the variable value is unchanged.

7. Locals window

Command: View/Locals

This window displays local variables of the current function or, more precisely, local variables of *the current watch* frame. for information about how to view frames see Virtually call stack unwinding.

You can use the Locals window like a Globals window [15]. Variable types are not marked, because the Locals window displays variables only.

8. Protocol window

Command: View/Protocol

The Debugger logs messages about errors and exceptions, and also informational messages in the protocol.

This window displays last added protocol lines.

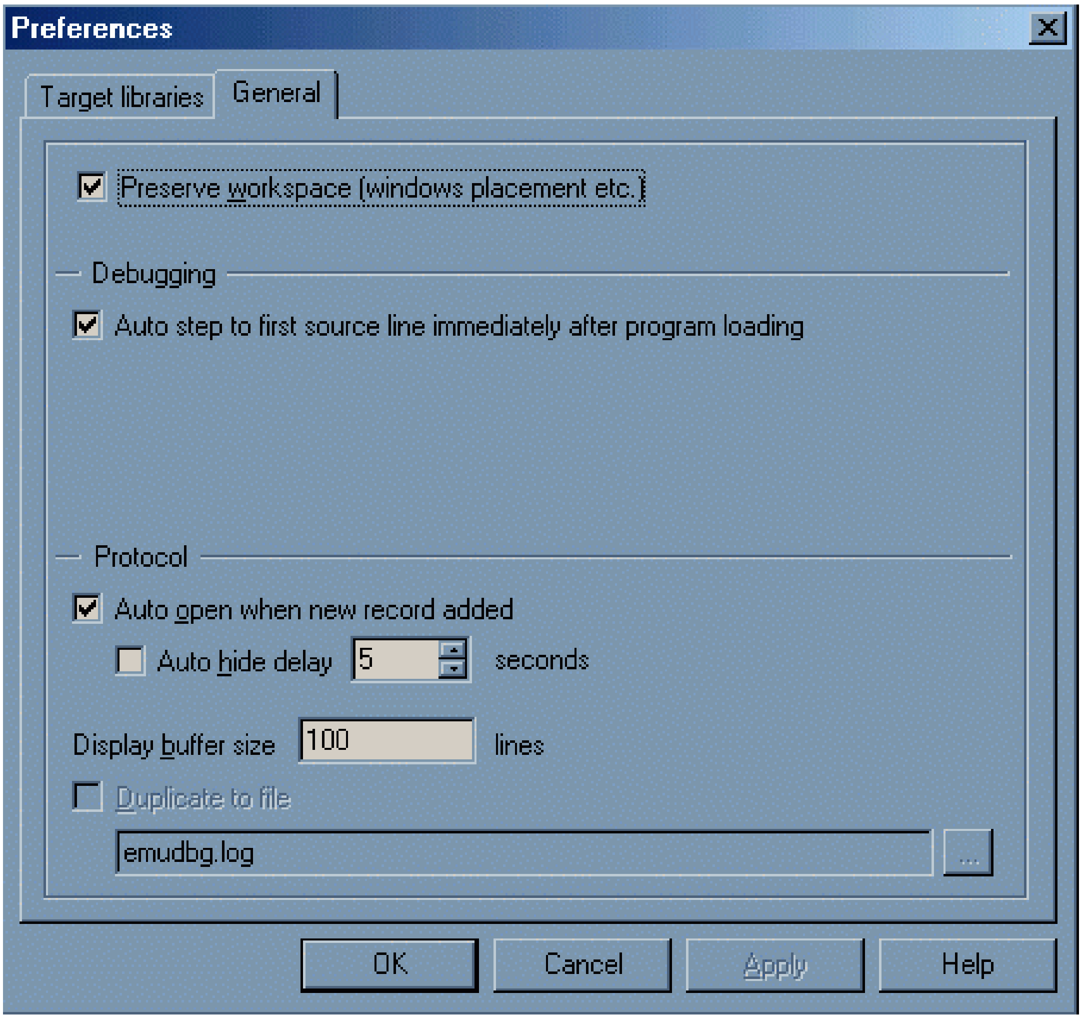
You can change protocol settings using General property sheet in the Preferences dialog box.

Chapter 6. The Debugger Preferences

The Edit/Preferences... menu commands allow you to adjust a number of settings that have a global effect on the behavior of the Debugger.

The Preferences dialog box contains two tabbed pages of settings.

1. General Settings Page



☑ Preserve workspace (windows placement etc.)

☑ Auto step to first source line immediately after program loading

☑ Auto open when new record added

☐ Auto hide delay 5 seconds

Display buffer size 100 lines

☐ Duplicate to file

emudbg.log

Preserve workspace (windows placement etc.)

When this option is selected, the debugger stores (and restores after start) windows placements and windows states (maximize/minimize). By default this option is **on**.

1.1. Protocol

Auto open when new record is added

The Debugger opens the protocol window automatically when a new protocol record is added. By default this option is **on**.

Auto hide delay is ... seconds

When this option is selected and the protocol window was opened automatically, the debugger hides the protocol window after the delay. By default this option is **on**.

The delay value can be changed. By default this value is **5 seconds**.

Display buffer size

Use this control to define the number of displayed lines in the protocol window. By default this value is **100 lines**.

Duplicate to file

Use this option to save the protocol in a file. By default this option is **off**.

Report file name can be changed. By default this name is **emudbg.log**.

1.2. Target libraries page

Using this page you can manage registered target libraries.

SDK installer registers all target libraries included in the SDK package, so generally you are not required to use this page.

See Target library registration for details.