

Project Debugging with MDK-ARM

Notes:

- This document assumes the use of MDK-ARM Version 5.xx (*µVision5*), installed with the STM32L1xx device family pack and ST-Link USB driver, and that a project has been created with the *µVision5* IDE, and a debug session initiated to download the project, as described in the document *STM32L100C-Discovery Board Projects*.

The MDK-ARM debug window (Figure 1) is the same whether debugging in the target hardware or the simulator. The debugger allows you to run/stop/step the program, use breakpoints, and to monitor selected program elements and microcontroller resources.

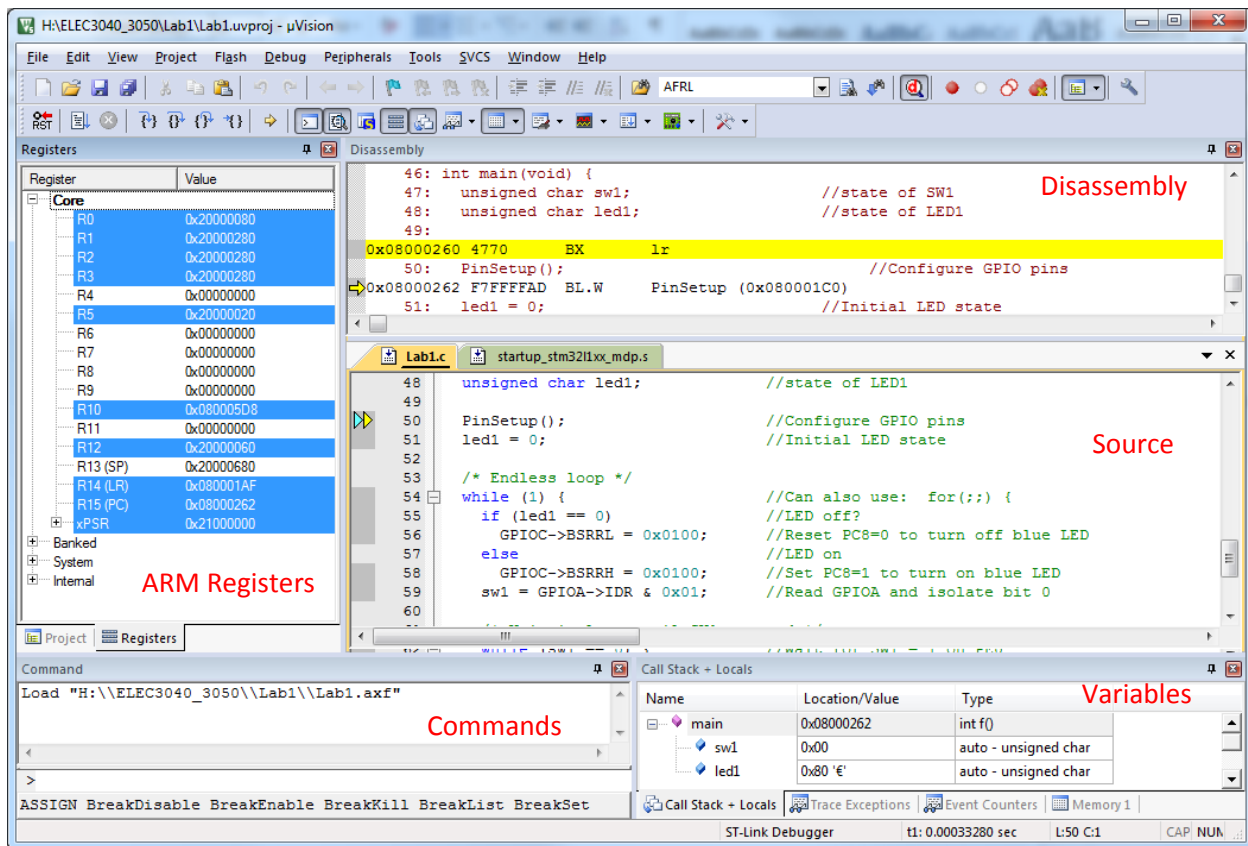


Figure 1. MDK uVision5 debug window.

- Click the **Debug** icon in the *µVision5* IDE, or select **Debug** → **Start/Stop Debug Session** to download the project code to the target board, program it into the microcontroller's flash memory, and open the Debug Window.
- Click on the **Debug** icon in the Debug window to stop the debug session and reenter the *uVision5* IDE.

Upon exiting a debug session, all Debug window settings will automatically be saved. These settings will be restored the next time you initiate a debug session for this project.

Source and Disassembly Windows

The **Source** window, in the center of the debug window (see Figure 1), displays the program code in selected project files, in the language used in each file (C, assembly, etc.) Clicking on a tab at the top of the window displays the source code in that file. If you wish to view a source file that does not have a tab in this window, you may add it from the menu bar by selecting **File > Open**, and then selecting the desired file.

The **Disassembly** window, immediately above the Source window, displays the disassembled code corresponding to the file currently displayed in the Source window. If the source is a C file, the disassembled code shows the assembly language generated for each C statement by the C compiler, including the hexadecimal memory address and object code for each assembly language statement.

In both windows, a yellow arrow or marker points to the next instruction to be executed. A blue marker in the Source window indicates a “cursor” position, which can be used during debugging. Shaded boxes at the left edge of these panes may be used to insert and remove breakpoints, as described below.

Running the program

A program may be executed one instruction at a time, executed up to a breakpoint, or simply executed without halting. Execution of the program is controlled via the icons in the left portion of the tool bar, immediately above the Register window. (These operations can also be executed from the Debug menu in the menu bar or from a pop-up menu produced by right-clicking in a debug window.) From left to right, the icon functions are as follows.

- **Reset** – reset the CPU and wait for the program to be started.
- **Run** – begin executing the program and continue until some stop condition is reached, such as a breakpoint or error condition
- **Stop** – stop execution of the program, with all panes indicating the state of the program and CPU at the time the stop took effect
- **Step** – execute a single instruction of the current program (C or assembly)
- **Step Over** – similar to Step, but execute any function/subroutine as a single “step”
- **Step Out** – executed instructions until the current function/subroutine is exited
- **Run to Cursor** – you may click on any instruction to set the position of a cursor (indicated by a blue marker at the left edge of the Source pane), and then execute instructions until that instruction is reached (at this point, the blue and yellow markers will coincide.)

Breakpoints

A “breakpoint” is a designated instruction at which program execution should stop (breaking the flow of the program). Breakpoints are useful debugging tools

- Determine whether the program reaches that point in the program. If the program never stops, then one knows that the instruction at the breakpoint was never executed.
- Stop the program to examine and/or change program variables, memory, or microcontroller registers at that point in the program.
- Stop the program to allow one to step through instructions individually to investigate the flow of the program one instruction at a time.

A breakpoint can be set by clicking in the shaded box at the left edge of the Source or Disassembly pane, next to the instruction at which the program should stop. A red dot will appear to mark the location of the breakpoint. Clicking on a red dot will remove the breakpoint.

One can also set and remove breakpoints via the Debug menu in the menu bar, by right-clicking in the Source or Disassembly pane to produce a pop-up menu, or by clicking on breakpoint icons in the top left tool bar of the debug window. Options available are:

- Insert/remove breakpoint at the current cursor position.
- Kill all breakpoints in a program. This is often more convenient than removing breakpoints one at a time.
- Enable/disable breakpoint at the current cursor position. A defined breakpoint can be “disabled” to prevent the program from stopping at that instruction, and then subsequently “enabled” to allow the program to break at that instruction.

Monitoring program variables and system resources

There are several debug windows that display useful information during program execution, including program variables, CPU registers, system memory, and various microcontroller peripheral function registers. These windows will update dynamically during execution of a program, so that one may determine the state of the program and whether the program is executing as expected.

Register Window (left side of the debug window) - displays the current contents of the CPU registers.

Call Stack + Locals (bottom right corner of the debug window) – displays the current stack contents, including:

- The names of the main program and any “called” functions. For example, if the “main” program calls function X, which then calls function Y, then all three functions will be listed in this window, including return addresses and any local variables within each function.

- Local variables within the currently-executing functions. Local variables are allocated memory on the stack when a function is entered, and then removed from the stack when the function returns to the calling program. You may right-click on any variable to change the display format, for example hexadecimal vs. decimal format.

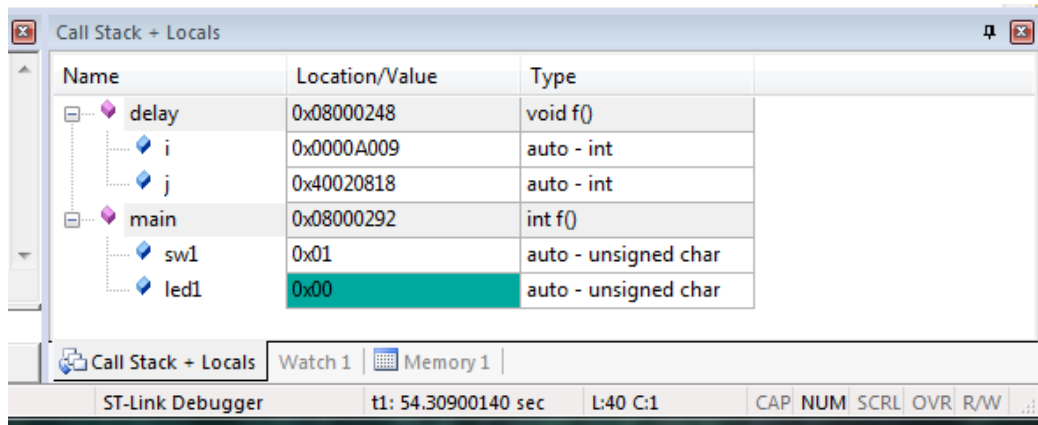


Figure 2. Call Stack + Locals Window. Here the *main* program has called function *delay()*, with each having two local variables.

Memory Window (bottom left corner of the debug window) – Up to four memory windows can be opened, each displaying the contents of selected addresses in memory. As shown in Figure 3, the starting address of a block of memory to be displayed is entered in the Address box, remembering that hexadecimal values begin with “0x” (otherwise the address will be interpreted as a decimal value.) The data format can be changed as desired, to facilitate studying the data, by right-clicking in the window and selecting the desired format:

- Unsigned or Signed number format.
- For each format, signed or unsigned, you can select “char”, “short”, “int”, or “long”, to display data as one, two, four, or eight-byte values, respectively. Figure 3 format is Unsigned > int.
- Decimal – change the displayed data from hexadecimal to decimal format.

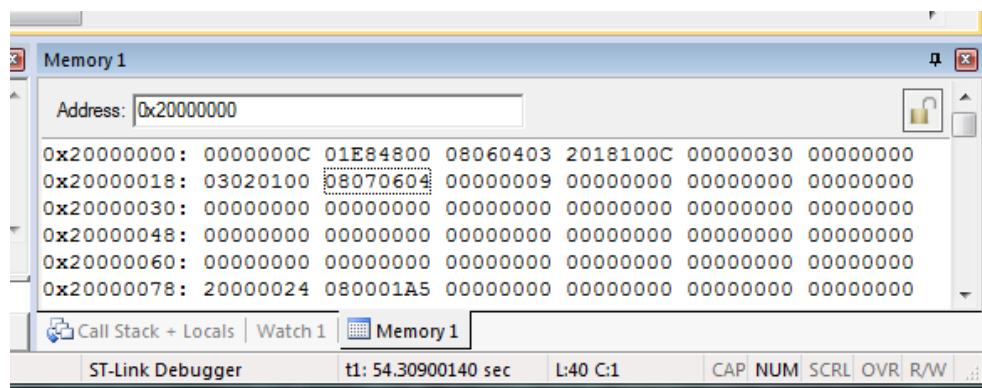


Figure 3. Memory 1 Window, showing the contents of memory beginning at address 0x20000000, displayed as 32-bit unsigned values.

Watch Window (bottom left corner of the debug window) – displays values of selected program variables and resources. This enables one to monitor the key program variables to determine the current state of the program, and whether the program is producing expected results. One or two Watch windows can be created during a debug session.

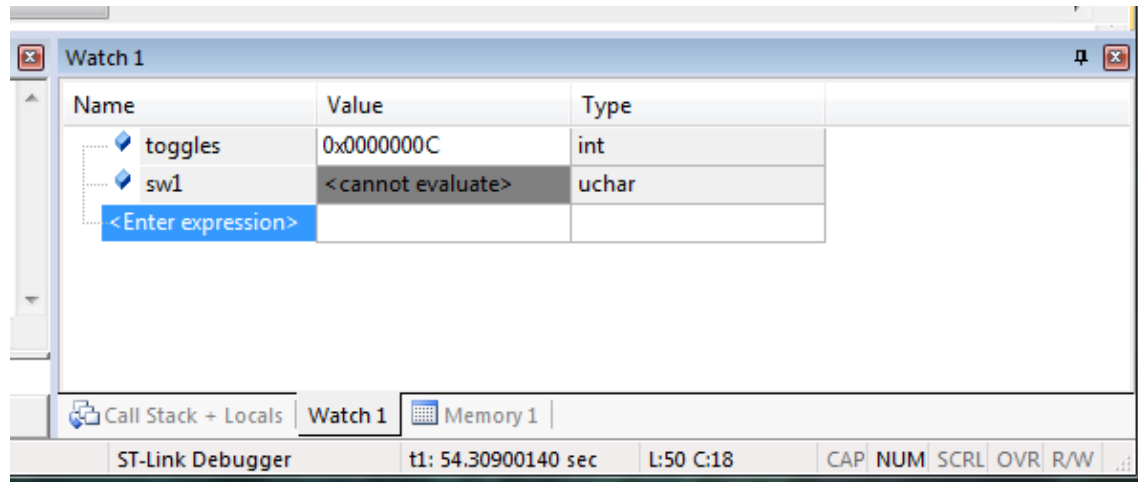


Figure 4. Watch 1 Window, showing the current value of global variable “toggles” and main program local variable “sw1” (undefined at the moment, since the program is executing another function.)

To add a variable, for example “toggles”, to window Watch 1, locate variable “toggles” in any statement in the Source Window, right click on it, and select: *Add ‘toggles’ to > Watch 1*. The variable will be displayed, as in Figure 4, with its current value and data type. You can change the format of the displayed value between hexadecimal and decimal by right-clicking on the variable name in the Watch window and selecting the desired format. The variable can be removed from a Watch window by right-clicking on the variable name in the Watch window and selecting *Remove Watch ‘toggles’*.

You can also change the current value of a variable by double-clicking on the value displayed in the Watch window and entering the desired value. This might be useful if you need to test a part of a program that is executed only for a particular value of some variable.

The values displayed in a Watch window change dynamically as the program executes, for variables that are “within scope”. Referring to Figure 4, variables “within scope” include any global variables in the program, such as variable “toggles”, and any local variables within the currently-executing function. In Figure 4, variable “sw1” is defined in a different function from the one that is currently executing, and thus the value is listed as “cannot evaluate”.

Other Debug Windows: can be opened from the View menu in the Debug Window menu bar.