Project Debugging with the *µVision* Debugger

Notes:

- This document assumes ARM Keil MDK (*µVision IDE Version 5.xx*) is installed with the required ST-Link USB driver and device family pack for the target board:
  STM32F4xx_DFP for STM32F411E-Discovery board
  STM32F3xx_DFP for STM32F3348-Discovery board
- It is also assumed that a project has been created and successfully built with the *µVision* IDE, and a debug session initiated to download the project, as described in the document *STM32 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.

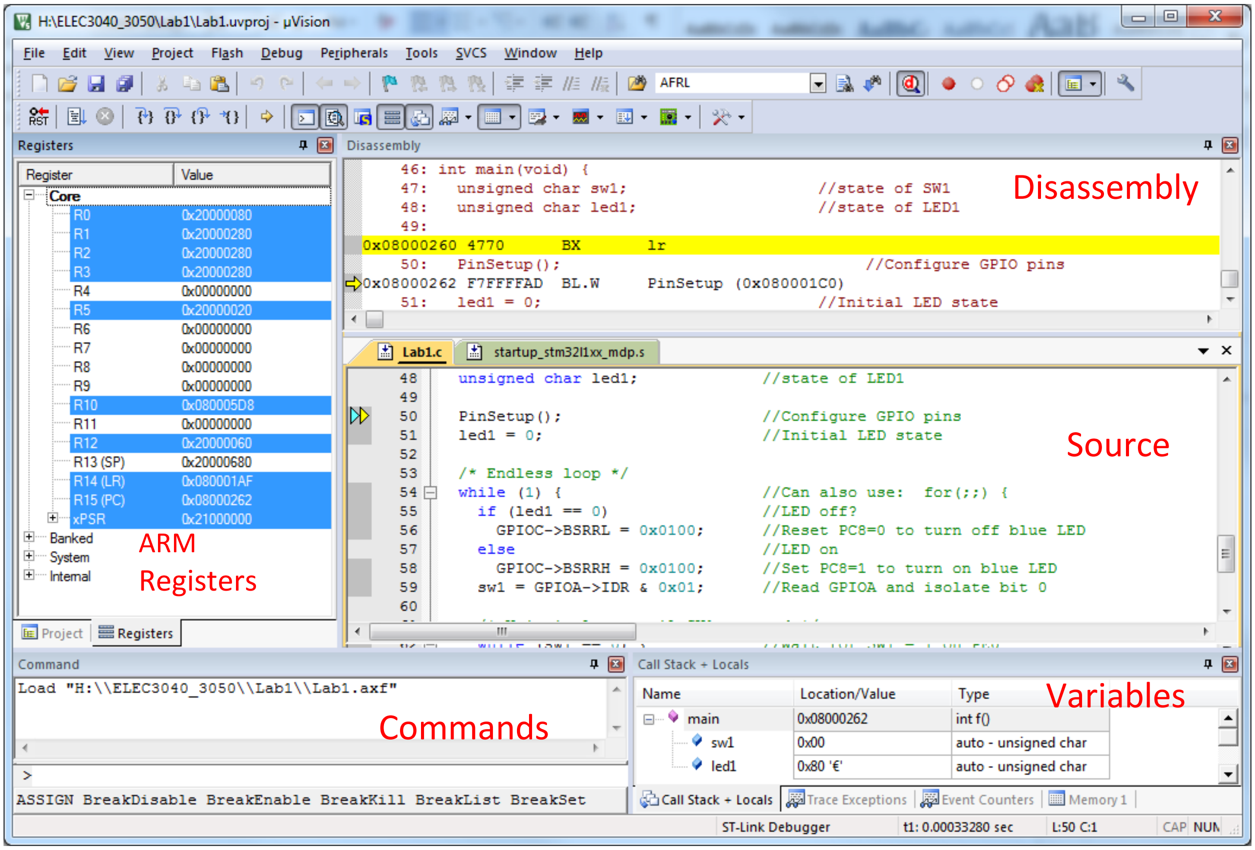

Figure 1. MDK uVision debug window.

- Click the ***Debug*** icon in the *µVision* IDE or select ***Debug → Start/Stop Debug Session*** to download the project code to the target board (or simulator), 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 return to the *uVision* 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.

Notes:

- If the arrow in the disassembly window does not point to one of your program instructions, the program counter (PC) register has not been set to the first address of your program in memory. This must be changed with a debug command (see Section 8 of the *STM32 Discovery Board Projects* document) or by double-clicking on the PC register in the ARM Registers pane and changing the value to the correct address.
- The shaded boxes at the left edge of the Source pane indicate executable instructions. If there is no shaded box where you want to set a breakpoint, then that instruction did not assemble correctly and there is no corresponding instruction code in memory.

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. (As indicated above, if there is no shaded box next to an instruction, then that instruction did not assemble and there is no instruction code in memory.)

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 may be 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.

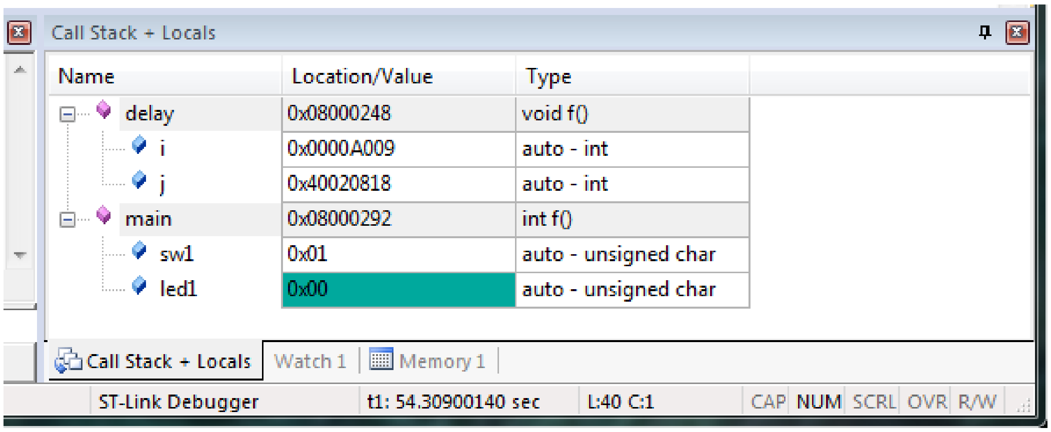

Figure 2. Call Stack + Locals Window. Here the *main* program has called function *delay()*, with each havng two local variables.

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(a), 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.

See different example formats in Figures 3(b)-3(d). You should select a format that best facilitates studying the information displayed in the window. If necessary, you may open multiple memory windows with a different format for each.

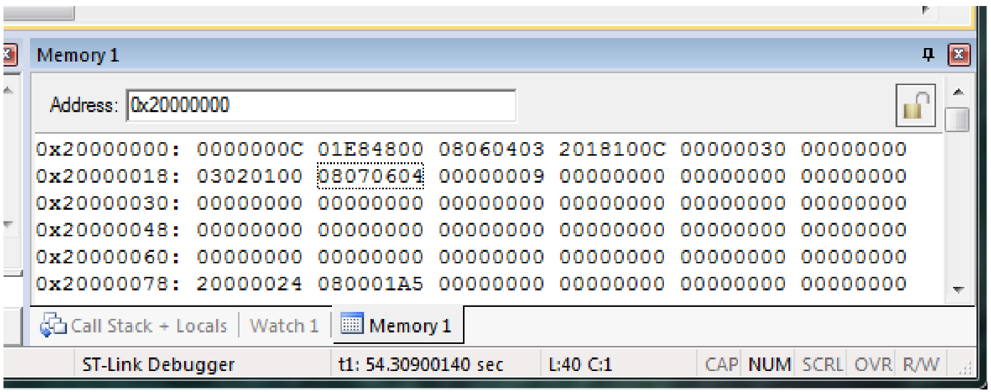

Figure 3(a). Memory 1 Window, showing the contents of memory beginning at address 0x20000000, displayed as 32-bit unsigned integer values.

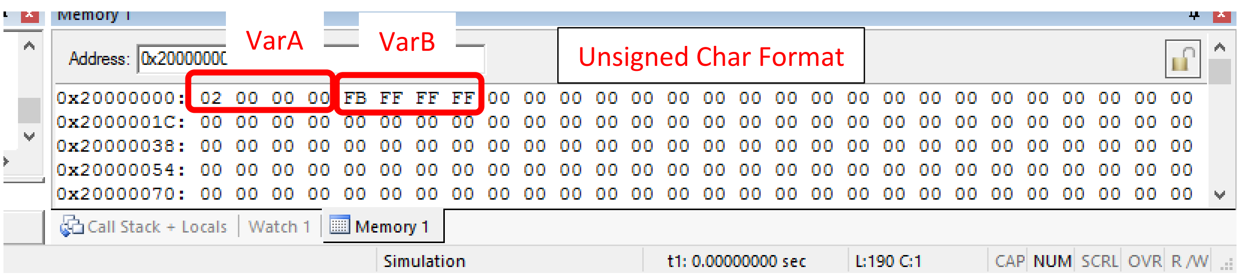

Figure 3(b). Memory 1 Window, showing values in “Unsigned Char” (8-bit byte) format.

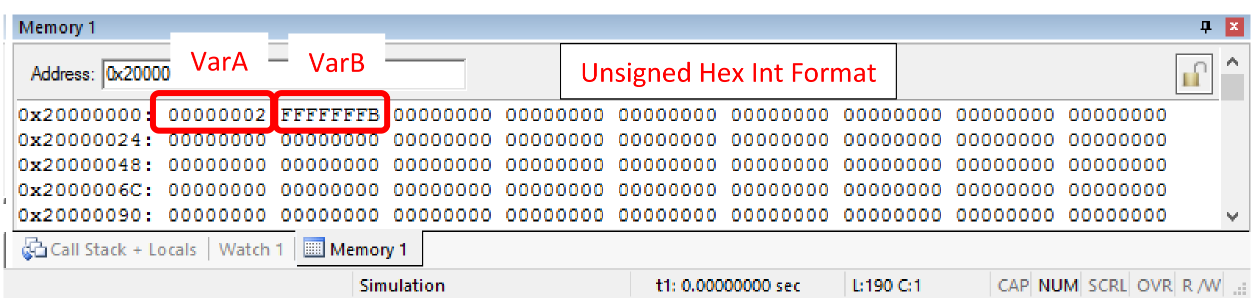

Figure 3(c). Memory 1 Window, showing values in “Unsigned Hex Int” (32-bit word) format.

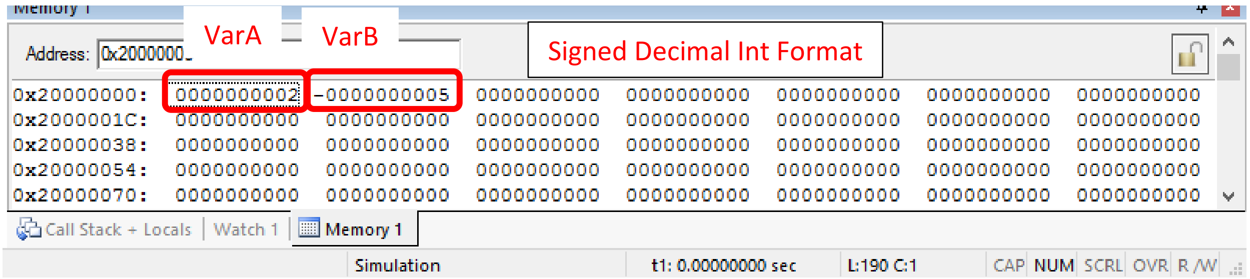

Figure 3(d). Memory 1 Window, showing values in "Signed Decimal Int" (32-bit word) format.

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. To display a variable in a watch window, a variable must be "global" and not "local". For assembly language, refer to the EXPORT directive described below.

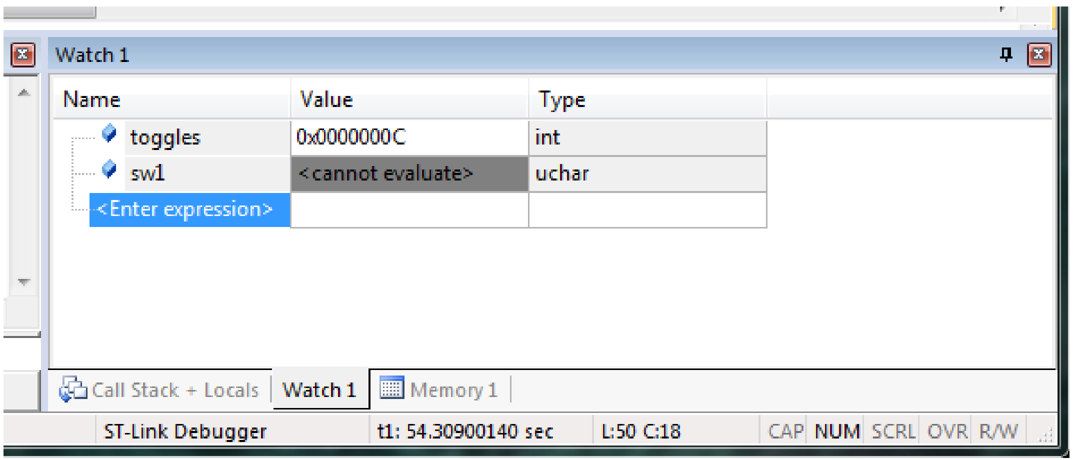

Figure 4. Watch 1 Window, showing the current value of global variable "toggles" and main program local variable "sw1" (undefined at the moment, since the program is executing another function.)

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'.*

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".

Assembly language: To display data in a watch window (as in Figure 4), corresponding to memory labels in a data area, the labels must be “exported” to make them global. For example:

```
        EXPORT toggles        ;export label "toggles" to make it "global"
        EXPORT sw1            ;export label "sw1" to make it "global"
toggles dcd     value         ;32-bit data
sw1     dcb     value         ;8-bit data
```

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.

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

Logic Analyzer Window (Not supported by the STM32F3348-Discovery board): graphically displays values of selected global variables over time. The procedure for setting up the logic anlyzer is as follows:

1. **Configure Serial Wire Viewer (SWV):**
   a. Select **Target Options > Debug tab > Settings**. On the right side of this window. Confirm SW is selected. SW selection is mandatory for SWV. ST-Link uses only SWD.
   b. Select the Trace tab, shown in Figure 5. Select ***Trace Enable***. Unselect ***Periodic*** and ***EXCTRC***. Set ***Core Clock*** to the frequency that you have set for the core of your microcontroller. (168 MHz was used for the project shown in Figure 5.) Click OK to return to the Debug tab.

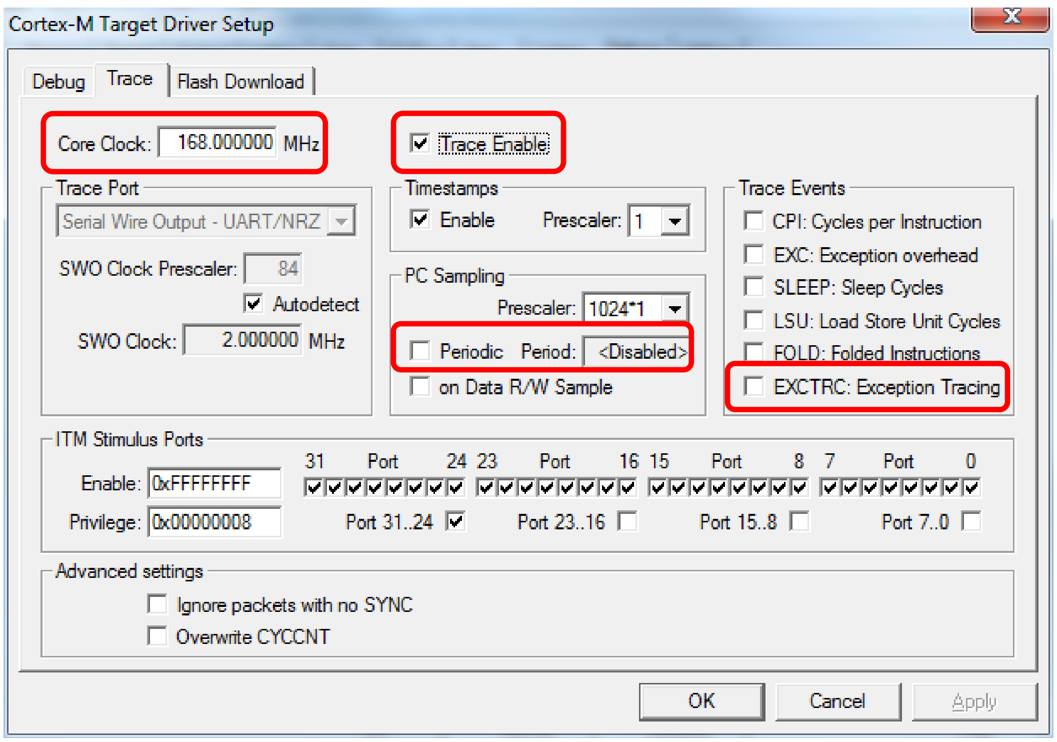

Figure 5. Target options to configure the logic analyzer tool.

2. Create a debug initialization file (ex. ***STM32_SWO.ini***) containing the following lines or add them to an existing debug initialization file. Enter that file name in the Debug tab as shown in Figure 6. This configures the STM32 SWV module and default is for SWV. Click OK to return to the main menu.

```
FUNC void DebugSetup (void) {
// <h> Debug MCU Configuration
//   <o1.0>    DBG_SLEEP     <i> Debug Sleep Mode
//   <o1.1>    DBG_STOP      <i> Debug Stop Mode
//   <o1.2>    DBG_STANDBY   <i> Debug Standby Mode
//   <o1.5>    TRACE_IOEN    <i> Trace I/O Enable
//   <o1.6..7> TRACE_MODE    <i> Trace Mode
//             <0=> Asynchronous
//             <1=> Synchronous: TRACEDATA Size 1
//             <2=> Synchronous: TRACEDATA Size 2
//             <3=> Synchronous: TRACEDATA Size 4
//   <o1.8>    DBG_IWDG_STOP <i> Independant Watchdog Stopped when Core is halted
//   <o1.9>    DBG_WWDG_STOP <i> Window Watchdog Stopped when Core is halted
//   <o1.10>   DBG_TIM1_STOP <i> Timer 1 Stopped when Core is halted
//   <o1.11>   DBG_TIM2_STOP <i> Timer 2 Stopped when Core is halted
//   <o1.12>   DBG_TIM3_STOP <i> Timer 3 Stopped when Core is halted
//   <o1.13>   DBG_TIM4_STOP <i> Timer 4 Stopped when Core is halted
//   <o1.14>   DBG_CAN_STOP  <i> CAN Stopped when Core is halted
// </h>
  _WDWORD(0xE0042004, 0x00004027);  // DBGMCU_CR
  }
```

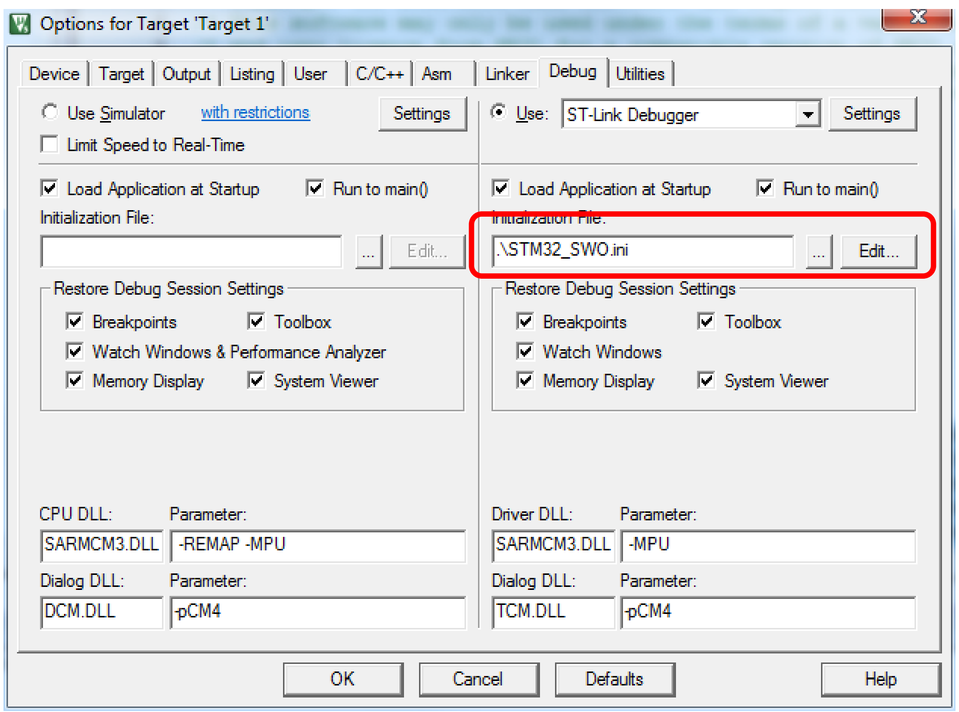

Figure 6. Debug initialization file configures SWV for the logic analyzer.

3. To open and configure the logic analyzer, within the Debugger open View/Analysis Windows and select ***Logic Analyzer***, or select the LA window on the toolbar. The LA can also be configured while the program is running.

4. In your source file, right click on a variable to be displayed (ex. **value)** and select *Add value to*… and then select *Logic Analyzer*. You can also Drag and Drop or enter manually. Note that this should be a "global variable", so that it always has a value.

5. Click on the Select box and the LA Setup window appears (Figure 7). With **value** selected, set Display Range Max: to 0x15 as shown in Figure 7, and then click on Close.

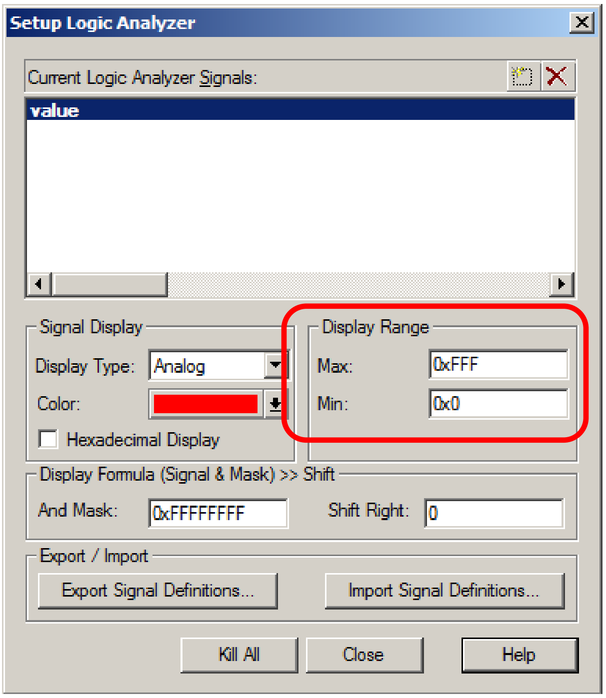

Figure 7. Logic Analyzer setup window – configure variable display range.

6. Click on Run to run the program. In the Logic Analyzer window (Figure 8), click on Zoom Out until Grid is about 1 second. The variable value will increment to 0x10 (decimal 16) and then is set to 0.

   TIP: You can show up to 4 variables in the Logic Analyzer. These variables must be global, static or raw addresses such as *((unsigned long *)0x20000000).

7. Enter the static variable **btns** into the LA and set the Display Range Max: to 0x2. Click on RUN and press the User button and see the voltages in Figure 8.

8. Select ***Signal Info***, ***Show Cycles***, ***Amplitude*** and ***Cursor*** to see the effects they have.

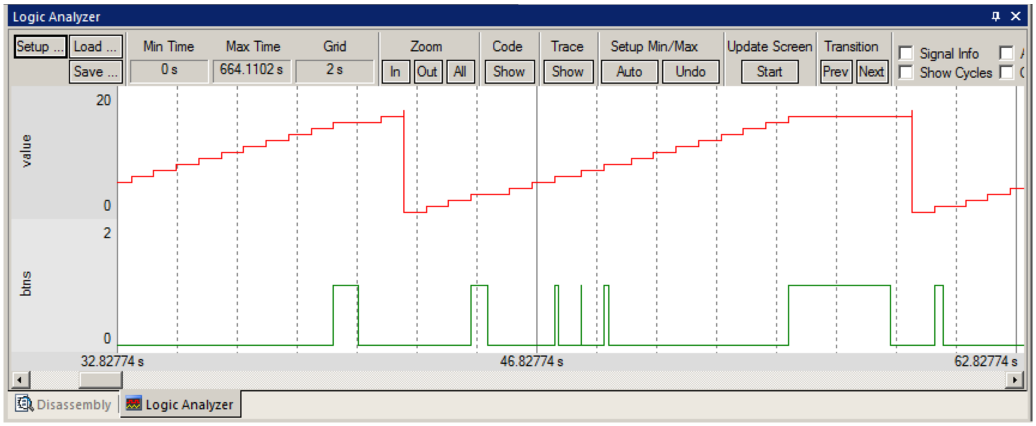

Figure 8. Logic Analyzer showing two variables.

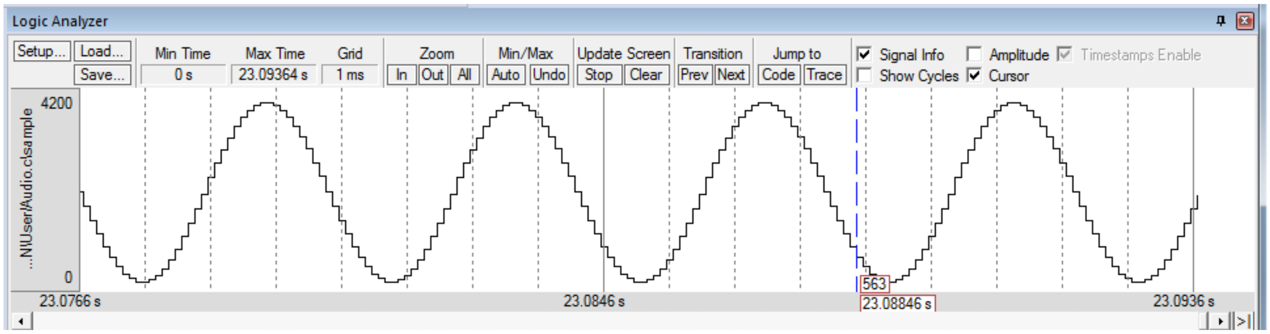

Figure 9.

Example

Global variables **phasea** through **phased** of Figure 10 are toggled between 0 and 1 at different rates by tasks of a “blinky” program.

1. Add the four variables to the Logic Analyzer window. These variables will be listed on the left side of the LA window as shown in Figure 11.

   Note: The Logic Analyzer can display static and global variables, structures and arrays. It can’t see locals: just make them static. To see peripheral registers read or write to them and enter them in the LA. Note that you can view signals that exist mathematically in a variable and not available for measuring in the outside world.

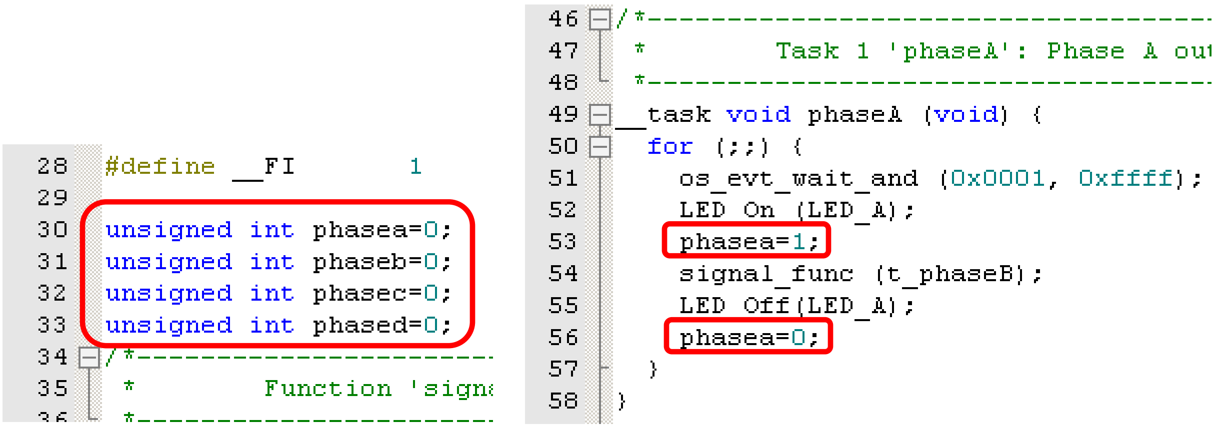

Figure 10. Four global variables to be monitored in the Logic Analyzer.

2. Adjust the scaling according to maximum/minimum values to be displayed. Click on the LA *Setup* icon and click on each of the four variables and set Max. in the Display Range: to 0x3. (In this application, variables toggle between 0 and 1.)
3. Use the *All*, *OUT* and *In* buttons set the range to 1 second or so. Move the scrolling bar to the far right if needed.
4. As shown in Figure 11, select *Signal Info* and *Show Cycles*. Click to mark a place and move the cursor to get timings. Place the cursor on one of the waveforms and get timing and other information as shown in the inserted box labeled ***phasec***.

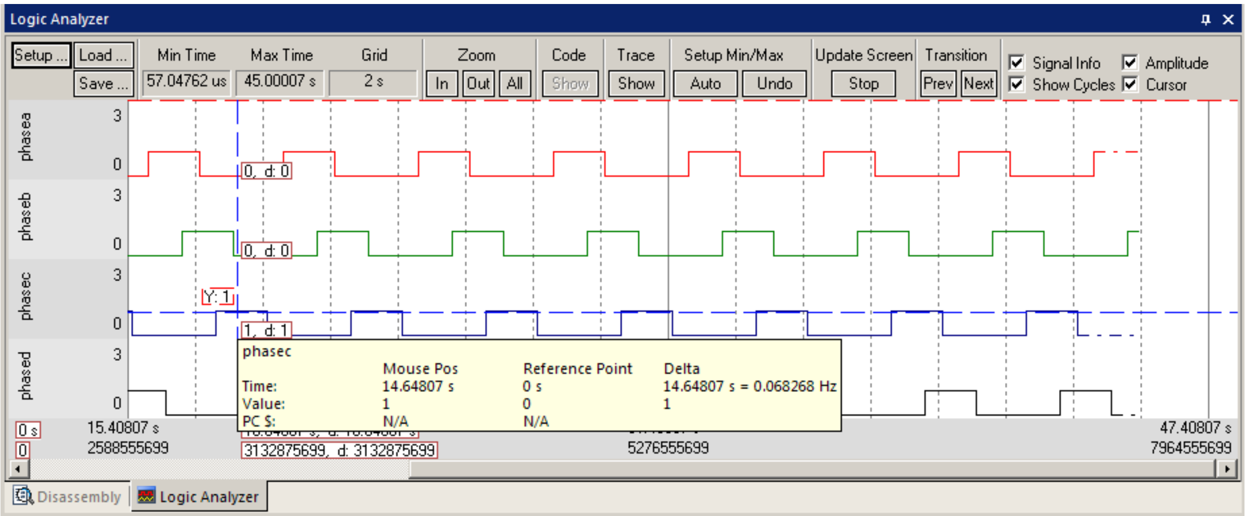

Figure 11. Logic Analyzer measurements of signal *phasec*.