

Proteus VSM and Keil Development Tools

Introduction

This document contains instructions for configuring Proteus VSM to work with the Keil IDE. It also walks through some basic debugging techniques and provides a short tutorial on debugging a simulated VSM circuit with the Keil IDE. It is not intended in any way to provide a complete list of the features of either product but rather is meant to supply some specific information on the interoperability of the two applications.

Getting Started

For the purposes of the following discussion we will assume that both Proteus VSM and the Keil software has been installed in the default directories on your computer. You will be able to follow the subsequent steps with demonstration versions of both packages although you will find that you need to purchase the following VSM components if you want to simulate your own designs.

- ? Proteus VSM Professional (Schematic Capture and Simulation Engine).
- ? MCS8051/52 Microcontroller Family.
- ? Peripherals Library.

Should you wish to write programs greater than 2K in size you will also need to purchase a Professional version of the Keil IDE.

How does it work?

Interaction between Proteus and the Keil IDE is achieved via the *Virtual Debug Monitor (VDM) Interface*. This interface defines a mechanism whereby the Keil IDE can control a VSM simulation session in much the same way as it might control an in-circuit emulator.

Actual communication is achieved through TCP/IP. This has the advantage that a debugging session can be run either on one computer or on two computers without the need for any external hardware other than a typical office network.

How do I set it up?

Although a lot of the hard work is done for you, some manual configuration is still necessary for successful communication. A step by step guide is given below.

- ? Ensure that TCP/IP networking is installed on the computer(s) you are going to be using. Almost all users will have this installed by default but it is easily checked. Look for TCP/IP under the following path:

Control Panel – Network – Protocols (Windows NT) or

Control Panel – Network – Configuration (Windows 95/98)

- ? Copy the VDM51.dll file from the MODELS directory of your Proteus VSM Installation to the BIN directory of your Keil installation. Typically these two directories are located at:

C:\Program Files\Labcenter Electronics\Proteus 5.2 Professional\MODELS

and

C:\KEIL\C51\BIN

- ? Using NOTEPAD (or your favourite text editor), edit the TOOLS.INI file that is located in the directory in which you have installed the Keil IDE. The full path to this file is typically C:\Keil\Tools.ini.

Towards the bottom of this file under the section [C51] you will find the line

TDRV0=BIN\MON51.DLL ("Keil Monitor-51 Driver")

Immediately below this add a line that specifies the location of the copied VDM51.DLL. By default this will be

TDRV1=BIN\VDM51.DLL ("Proteus VSM Monitor-51 Driver")

This done, you can now save the TOOLS.INI file.

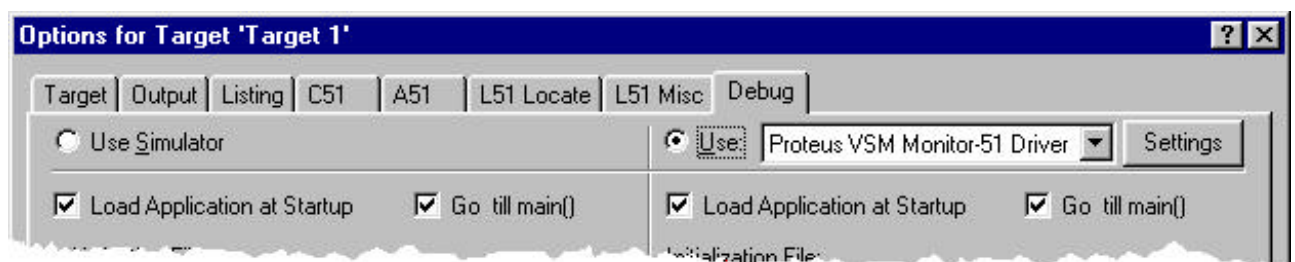
Note: If you have the Keil application open you will need to close it down before the above changes will take effect.

You should now have configured the Keil IDE to accept the VDM driver allowing communication between the applications. The next step is to prepare a Keil project for debugging under Proteus VSM.

Preparing a Debug Session

For the purposes of this documentation we will use the C51 calculator project as provided in the samples directory of the Proteus installation. A guide to configuring the applications is given below:

- ? Start up ISIS and load the Keil Calculator sample design. Typically this can be found in
C:\Program Files\Labcenter Electronics\Proteus 5.2 Professional\Samples\C51 Calculator\
Note that if you have the demonstration version of the Proteus software any alteration to the design will disable the simulation functionality.
- ? Select the *Use Remote Debug Monitor* option from the *Debug* menu. A message to the effect that the Virtual Debug Monitor is enabled should appear on the status line (at the bottom of the application window). If you get error messages it is most likely that there is a problem with your TCP/IP configuration.
- ? Start up the Keil environment and load in the project CALC.UV2 which, by default, can be found via the above path. If you are using the demonstration version of the Keil software note that it is limited to 2K code. Should you decide to change the code and exceed this limitation the program will not compile.
- ? Invoke the *Options for Target 'Target 1'* command from the *Project* Menu.
- ? Select the *Debug* tab of this dialogue form and change the selection from the *Use Simulator* option to the *Use Keil Monitor-51 Driver* option on the right hand side of the dialogue form.
- ? Change the combo-box to select the *Proteus VSM Monitor-51 Driver* option. If you do not have this option please refer to the instructions above for setting up the installations.
- ? If you intend on running ISIS on a second computer, click the *Settings* button and enter the IP address or DNS name of the target computer into the *Host IP Address* field.
- ? In most circumstances (and for the current demonstration) you should select *Load Application at Startup* and *Go until Main*. Selecting the former means that it is not necessary to specify the program file for the 8051 model in ISIS as UV2 will load the program into the model at the start of each debugging session.



The Keil configuration Dialogue form showing the correct setup for debugging in conjunction with Proteus VSM.

You should now have a project loaded in Keil which is configured to control a debugging session on the circuit loaded in ISIS.

Debugging the circuit.

Starting a Debug Session

The first task here is to set a breakpoint in the program code from within the Keil IDE. A suitable point would be the entrance to the LCD initialisation routine so select *calc.c* from the left hand pane and place the mouse cursor on the line with the code:

```
initialise(); // Initialize the LCD
```

From the *Debug* menu select the *Insert/Remove Breakpoint* command – you should now see a red rectangle at the start of the line which represents the breakpoint.

Note: If you want to specify a shortcut key for this or any other commonly used functions you do so via the Options command on the View menu.

Now select the *Start/Stop Debug Session* option from the *Debug* menu. This will initialise the debug session and should have two noticeable effects.

1. The main Keil window will change and display an assembly language listing. A small yellow arrow on the left hand side will indicate the current instruction as dictated by the breakpoint. The left hand pane of the Keil window will change to display the register bank and system registers.
2. Simulation will start in ISIS and then pause prior to execution of the instruction specified by the breakpoint. The status bar at the bottom of the application window will display the amount of time that the simulation has run and the PC value when the breakpoint is reached.

Note: If you are running both applications on the same computer you can toggle between them using ALT+TAB.

Additional Debug Information

The *Peripherals* menu in the Keil IDE allows you to view the status and values of the Interrupt System, Timers, IO Ports, A/D Converter and the Serial Peripheral Interface. Additionally, you can look at the memory contents and use the Watch Window via the *View menu*.

ISIS provides a series of popup windows accessible from the *Debug* menu which allow you to view memory contents register values as well as use the configurable Watch Window. The values displayed in these should of course be identical to those in the Keil environment at any stage of the simulation and as such you can use either or both according to your personal preferences.

Another useful feature of ISIS is that you can visually display the logic states of the pins throughout the simulation. This option is accessible through the *Set Animation Options* command on the *Template* menu and provides an excellent snapshot of the state of the circuit.

Note: Both ISIS and Keil use highlighting to indicate that the value of an item in the watch window (or a register in the left pane of the Keil IDE) has changed since the last paused state.

Debugging Commands

The following commonly used debugging commands are available within the Keil environment.

Go

When paused this command will release the simulation back into free running mode.

Step

When paused this command will advance the simulation by one instruction.

Step Over

When paused this command will step over the current instruction and pause at the following instruction.

Step Out of Current Function

When paused this command will exit the current routine pausing at the instruction subsequent to the return.

Run to Cursor Line

When paused this command will release the simulation into free running mode until the line of code where the mouse cursor is currently positioned is reached.

Stop Running

When running this command will pause the simulation at the instruction currently being executed.

It is important to realise that these commands operate at the level of the 8051 assembly language and **not** at the level of the C source file. A *Step Over* command, for example, will not advance the simulation over the current line in the C file but rather step over the current instruction as shown in the disassembly window.

Stepping through the code

It is generally considered easier to control the debug process at the level of the program files rather than through the disassembly. Although this is again a personal choice we will work from our C program file and assume that users who prefer to work with the assembly language can extrapolate the principles involved.

You can view the `calc.c` file by selecting it from the *Window* menu. As we stand we can either step into the initialise routine or step over the routine and on to the next part of the program. For our purposes assume that we are not interested in this routine and so we want to run the simulation until we reach the entry point of the main function. We can do this in one of two ways:

- ? Set another breakpoint at the point to which we want to advance and then select the *Run* command from the *Debug* menu.
- ? Place the cursor on the line to which we wish to advance and select the *Run to Cursor Line* command from the *Debug* menu.

Using one of these methods advance the simulation over the initialise and output functions coming to halt at the line reading:

```
calc_evaluate();           (line 29)
```

Looking at ISIS we should now see some changes – the LCD display has initialized and the digit ‘0’ is displayed. Now let’s examine the calculator in action. We will set a breakpoint at the point of receiving a valid digit, i.e. on the line:

```
*bufferptr = key;         (line 52)
```

We can set the simulation running and use the keypad in ISIS to enter a digit. Do this, entering the digit ‘7’ on the keypad, and then view the disassembly in the Keil window.

We can immediately see from the red rectangle in the disassembly window that we have stopped at a breakpoint. This tells us that we have received a valid digit from the keypad. We will now take a slightly more in depth look at how this digit is passed through the program.

Disassembly in Detail

The first thing that happens in the disassembly is that we move the contents of `0x0B` and `0x0C` into the Data Pointer. Examination of the Internal Memory window in either ISIS or Keil reveals that this value is `0x0009`. Single step twice over these instructions and you should see that the value of the Data Pointer has indeed changed correctly.

Next we move the contents of `0x08` into the Accumulator. Again, investigation reveals that this value is `0x37` which is consistent with the result after stepping past the instruction.

Finally we move the contents of the Accumulator into the address specified by the Data Pointer. A further single step followed by an examination of the memory windows will confirm that this has happened. In essence we have moved the ASCII value for the digit seven into a memory location (which happens to serve as the first element of the specified array) which correctly reflects the current line in *Calc.c*.

The next line in our C file is a call to another routine taking a character pointer as an argument:

```
calc_display(number);      (line 53)
```

We can see from the disassembly that the next three instructions move values into registers R1-R3. These instructions are followed by a ‘long call’ instruction which clearly corresponds to our function call in the C program.

The Keil compiler by default passes generic pointers using three bytes, where the first byte specifies the memory type and the second and third bytes specify the high and low order bytes of the offset respectively.

In our case this translates to be the external data memory space with an offset of nine. Again, after single stepping, we can confirm from our memory windows that this location contains the value of our digit.

Note: The Keil C51.pdf detailing the operation of the compiler is available from the Books tab on the left hand pane of the Keil IDE or from the website.

We now have the choice of entering the routine via the *Step* command or advancing beyond the routine via the *Step Over* command. In order to follow our entered digit we should step into this routine.

We can see that the first thing that happens in our new routine is that the parameter passed in registers R1-R3 is stored freeing up these registers for local use.

Next we clear the accumulator and store it's value in memory locations 0x11 and 0x12. This corresponds to the execution of the first line in the `calc_display` function:

```
INT data i = 0;          (line 257)
```

Stepping through the disassembly we find ourselves at another 'long call' instruction. This corresponds to the call to the `clearscreen()` function in our C source file and, as such is not relevant to the current exercise. Use the *Step Over* instruction to advance the simulation beyond this instruction..

Note: Caution is required when stepping over functions during the debug of a real problem. Bear in mind that not only are you stepping over the execution of the function in question but also of any functions called by that function.

A quick check of the ISIS display at this point will confirm that the LCD display has indeed been cleared.

The next few instructions are quite involved and deal with the conditions specified in the C source file for allowing a character to be outputted to the display. Given that we know that our character is valid and will pass these checks we won't concern ourselves with them but simply single step through them.

The by now rather familiar action of loading registers R1-R3 with our parameter comes next before yet another function call – this time to the assembly language LCD controller routine.

At this point we will leave the trail and step over this function call (feel free to follow through the routines if you want). This has the immediate and rather gratifying effect that the digit seven is now correctly shown in the LCD display in ISIS.

Obviously there is a lot more to the debugging process than we have covered here but hopefully you should have some idea of the possibilities and advantages of using Proteus VSM and the Keil IDE in partnership to debug 8051 circuits. The next section of the document lists some other, less commonly used features of the products.

Additional Features

The Keil IDE allows you to enter a number of specific commands while debugging. These features are enabled via the *Command* tag on the *Output Window* (by default this is set on the *Build* tab). Options include writing values to ports, displaying memory ranges, assigning values to registers and several others. Integrated auto-complete functionality makes using this functionality fairly transparent.

Proteus VSM offers a number of useful visual animation options including *Logic State of Pins* and *Wire Voltage by colour*. These can be useful in providing a quick check of obvious problems before proceeding to debug in depth.