

PROTEUS

VSM

Virtual System Modelling

Incorporating PROSPICE - Berkeley SPICE3F5
with extensions for mixed mode simulation.

User Manual

Issue 6.0 - November 2002

© Labcenter Electronics

TABLE OF CONTENTS

INTRODUCTION.....	1
ABOUT PROTEUS VSM.....	1
ABOUT THE DOCUMENTATION.....	2
TUTORIALS.....	3
INTERACTIVE SIMULATION TUTORIAL.....	3
Introduction.....	3
Drawing the Circuit	4
Writing the Program	6
Debugging the Program.....	9
GRAPH BASED SIMULATION TUTORIAL.....	11
Introduction.....	11
Getting Started	11
Generators.....	13
Probes.....	13
Graphs.....	14
Simulation.....	16
Taking Measurements.....	17
Using Current Probes	18
Frequency Analysis	19
Swept Variable Analysis	20
Noise Analysis	21
INTERACTIVE SIMULATION	23
BASIC SKILLS.....	23
The Animation Control Panel	23
Indicators and Actuators	24
Setting Up an Interactive Simulation.....	24
ANIMATION EFFECTS.....	25
Overview	25
Pin Logic States	25
Show Wire Voltage as Colour.....	25
Show Wire Current as Arrows	25
ANIMATION TIMESTEP CONTROL.....	26
Overview	26
Frames Per Second	26
Timestep Per Frame	26
Single Step Time.....	26

HINTS AND TIPS	26
Circuit Timescale.....	26
Voltage Scaling.....	27
Earthing	27
High Impedance Points	27
VIRTUAL INSTRUMENTS.....	29
VOLTMETERS & AMMETERS.....	29
OSCILLOSCOPE.....	29
Overview	29
Using the Oscilloscope.....	30
LOGIC ANALYSER.....	32
Overview	32
Using the Logic Analyser.....	33
SIGNAL GENERATOR.....	34
Overview	34
Using the Signal Generator.....	34
DIGITAL PATTERN GENERATOR	36
Overview	36
Using the Pattern Generator.....	37
ThePattern Generator Component Pins.....	38
Clocking Modes	39
Trigger Modes	39
External Hold.....	42
Additional Functionality	42
VSM USER INTERFACE ELEMENTS	45
Rotary Dials	45
WORKING WITH MICROPROCESSORS.....	47
INTRODUCTION	47
SOURCE CODE CONTROL SYSTEM	47
Overview	47
Attaching Source Code to the Design.....	48
Working on your Source Code	48
Installing 3 rd Party Code Generation Tools	49
Using a MAKE Program.....	49
Using a 3 rd Party Source Code Editor	50
USING A 3 RD PARTY IDE.....	50
Using Proteus VSM as an External Debugger	51
Using Proteus VSM as a Virtual In-Circuit Emulator (ICE).....	52
POPUP WINDOWS	52

SOURCE LEVEL DEBUGGING WITHIN PROTEUS VSM	53
Overview	53
The Source Code Window.....	53
Single Stepping.....	53
Using Breakpoints.....	54
Variables Window	54
THE WATCH WINDOW	55
BREAKPOINT TRIGGER OBJECTS	55
Overview	55
Voltage Breakpoint Trigger – RTVBREAK.....	56
Current Breakpoint Trigger – RTIBREAK.....	56
Digital Breakpoint Trigger – RTDBREAK.....	56
GRAPH BASED SIMULATION	57
INTRODUCTION	57
SETTING UP A GRAPH BASED SIMULATION.....	57
Overview	57
Entering The Circuit	57
Placing Probes and Generators	57
Placing Graphs	58
Adding Traces To Graphs.....	58
The Simulation Process.....	59
GRAPH OBJECTS	59
Overview	59
The Current Graph.....	60
Placing Graphs	60
Editing Graphs	60
Adding Traces To A Graph.....	60
The Add Trace Command Dialogue Form	62
Editing Graph Traces	63
Changing the Order and/or Colour of Graph Traces	63
THE SIMULATION PROCESS.....	63
Demand Driven Simulation	63
Executing Simulations.....	64
What Happens When You Press the Space-Bar	65
ANALYSIS TYPES	67
INTRODUCTION	67
ANALOGUE TRANSIENT ANALYSIS.....	68
Overview	68
Method of computation.....	68

Using Analogue Graphs	69
Defining Analogue Trace Expressions.....	70
DIGITAL TRANSIENT ANALYSIS.....	71
Overview	71
Method of Computation.....	71
Using Digital Graphs.....	72
How Digital Data is Displayed.....	73
MIXED MODE TRANSIENT ANALYSIS.....	74
Overview	74
Method of Computation.....	74
Using Mixed Graphs.....	75
FREQUENCY ANALYSIS.....	75
Overview	75
Method of computation.....	75
Using Frequency Graphs	76
DC SWEEP ANALYSIS.....	77
Overview	77
Method of computation.....	77
Using DC Sweep Graphs.....	78
AC SWEEP ANALYSIS	79
Overview	79
Method of Computation	79
Using AC Sweep Graphs.....	79
DC TRANSFER CURVE ANALYSIS.....	80
Overview	80
Method of Computation.....	80
Using Transfer Graphs.....	80
NOISE ANALYSIS.....	81
Overview	81
Method of calculation	81
Using Noise Graphs.....	82
DISTORTION ANALYSIS.....	82
Overview	82
Method of Computation.....	83
Using Distortion Graphs.....	83
FOURIER ANALYSIS.....	84
Overview	84
Method of Calculation	84
Using Fourier Graphs.....	84
AUDIO ANALYSIS	85
Overview	85

Method of Calculation	85
Using Audio Graphs	85
INTERACTIVE ANALYSIS	85
Overview	86
Method of Computation	86
Using Interactive Graphs	86
DIGITAL CONFORMANCE ANALYSIS	87
Overview	87
Method of Computation	87
Using Conformance Graphs	88
DC OPERATING POINT	91
GENERATORS AND PROBES	93
GENERATORS	93
Overview	93
Placing Generators	94
Editing Generators	94
DC Generators	95
Sine Generators	95
Pulse Generators	96
Exponential Generators	97
Single Frequency FM Generators	97
Piece-wise Linear Generators	98
File Generators	99
Audio Generators	99
Digital Generators	100
PROBES	101
Overview	101
Placing Probes	101
Probe Settings	102
USING SPICE MODELS	103
OVERVIEW	103
USING A SPICE SUBCIRCUIT (SUBCKT DEFINITION)	104
USING A SPICE MODEL (MODEL CARD)	106
SPICE MODEL LIBRARIES	107
*SPICE SCRIPTS	108
HANDLING MODEL SIMULATION FAILURES	108
ADVANCED TOPICS	109
GROUND AND POWER RAILS	109

Why You Need a Ground.....	109
Power Rails	111
INITIAL CONDITIONS.....	112
Overview	112
Specifying Initial Conditions for a Net	112
Specifying Initial Conditions for Components	113
NS (NODESET) Properties	113
PRECHARGE Properties	114
TEMPERATURE MODELLING.....	114
THE DIGITAL SIMULATION PARADIGM	115
Nine State Model	115
The Undefined State.....	115
Floating Input Behaviour	116
Glitch Handling	116
MIXED MODE INTERFACE MODELS (ITFMOD).....	119
Overview	119
Using ITFMOD Properties	121
PERSISTENT MODEL DATA.....	121
TAPES AND PARTITIONING.....	122
Overview	122
Single Partition Operation.....	122
Tape Objects.....	123
Tape Modes	124
SIMULATOR CONTROL PROPERTIES.....	125
Overview	125
Tolerance Properties	125
Mosfet Properties	125
Iteration Properties	126
Temperature Properties	126
Digital Simulator Properties	127
TYPES OF SIMULATOR MODEL.....	128
How to tell if a Component Has a Model.....	128
Primitive Models.....	129
Schematic Models	129
VSM Models.....	130
SPICE Models.....	130
TROUBLESHOOTING.....	133
THE SIMULATION LOG.....	133
NETLISTING ERRORS	133
LINKING ERRORS	133

PARTITIONING ERRORS.....	134
SIMULATION ERRORS.....	134
CONVERGENCE PROBLEMS.....	135

INDEX	137
--------------------	------------

INTRODUCTION

ABOUT PROTEUS VSM

Traditionally, circuit simulation has been a non-interactive affair. In the early days, netlists were prepared by hand, and output consisted of reams of numbers. If you were lucky, you got a pseudo-graphical output plotted with asterisks to show the voltage and current waveforms.

More recently, schematic capture and on screen graphing have become the norm, but the simulation process is still non-interactive – you draw the circuit, press go, and then study the results in some kind of post processor. This is fine if the circuit you are testing is essentially static in its behaviour e.g. an oscillator which sits there and oscillates nicely at 1Mhz. However, if you are designing a burglar alarm, and want to find out what happens when a would-be burglar keys the wrong PIN into the keypad, the setting up required becomes quite impractical and one must resort to a physical prototype. This is a shame, as working ‘in cyberspace’ has so much to offer in terms of design productivity.

Only in educational circles has an attempt been made to present circuit simulation like real life electronics where it is possible to interact with the circuit whilst it is being simulated. The problem here has been that the animated component models have been hard coded into the program. Only limited numbers of simple devices such as switches, light bulbs, electric motors etc. have been offered, and these are of little use to the professional user. In addition, the quality of circuit simulation has often left much to be desired. For example, one major product of this type has no timing information within its digital models.

PROTEUS VSM brings you the best of both worlds. It combines a superb mixed mode circuit simulator based on the industry standard SPICE3F5 with animated component models. And it provides an architecture in which additional animated models may be created by anyone, including end users. Indeed, many types of animated model can be produced without resort to coding. Consequently PROTEUS VSM allows professional engineers to run interactive simulations of real designs, and to reap the rewards of this approach to circuit simulation.

And then, if that were not enough, we have created a range of simulator models for popular micro-controllers and a set of animated models for related peripheral devices such as LED and LCD displays, keypads, an RS232 terminal and more. Suddenly it is possible to simulate complete micro-controller systems and thus to develop the software for them without access to a physical prototype. In a world where time to market is becoming more and more important this is a real advantage.

It is also worth pointing out that the processing power of the modern PC is truly awesome. A 300MHz Pentium II PC can simulate simple micro-controller designs in real time, or even faster in some cases. And even where things slow down somewhat, the response time is more often

that not useable for software development. If you are serious about this game, you can go out and buy a 2GHz dual processor PC, which is far, far faster. This, then, debunks the other obvious objection to interactive simulation – that it would not be fast enough.

ABOUT THE DOCUMENTATION

This manual is intended to complement the information provided in the on-line help. Whereas the manual contains background information and tutorials, the help provides context sensitive information related to specific icons, commands and dialog forms. Help on most objects in the user interface can be obtained by pointing with the mouse and pressing F1.

PROTEUS VSM can be used in two rather distinct ways – either for *Interactive Simulation* or for *Graph Based Simulation* and this is reflected in the structure of the manual. Typically, you will use interactive simulation to see if a design works at all, and graph based simulation to investigate why it doesn't or to take very detailed measurements. However, there are no hard and fast rules. Some elements of the system, such as *Generators* are relevant to both modes of use and are given their own chapters for this reason.

Detailed, step by step tutorials are provided which take you through both types of simulation exercise. We strongly recommend that you work through these as they will demonstrate all the basic techniques required to get going with the software.

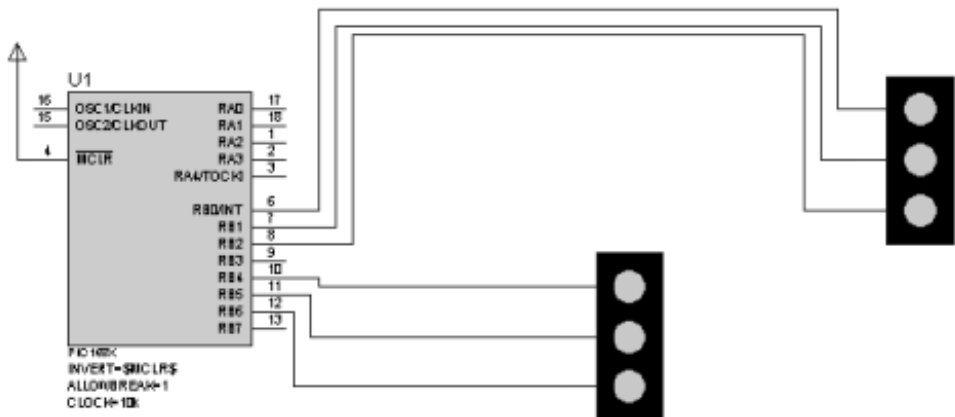
Information on creating VSM models is provided separately in the *VSM Software Development Kit (SDK)*, an on-line resource which is installed as standard with the professional version of Proteus.

INTERACTIVE SIMULATION TUTORIAL

Introduction

The purpose of this tutorial is to show you, through the creation of a simple schematic, how to conduct an interactive simulation using Proteus VSM. While we will concentrate on the use of Active Components and the debugging facilities of the ISIS Editor we will also look at the basics of laying out a schematic and general circuit management. Full coverage of these topics can be found in the ISIS Manual.

The circuit we will be using for the simulation is a pair of traffic lights connected to a PIC16F84 microcontroller as shown below.



Whilst we will be drawing the schematic from scratch, the completed version can be found as "Samples\Tutorials\Traffic.DSN" within your Proteus installation. Users who are familiar with the general operating procedures in ISIS may choose to use this ready made design and move on to the section on the microcontroller program. Please note, however, that this design file contains a deliberate error - read on for more information.

If you are unfamiliar with ISIS, the interface and basic usage are discussed at length in *A Guided Tour of the ISIS Editor* in the ISIS manual. and although we will touch on these

issues in the following section you should take the time to familiarise yourself with the program before proceeding .

Drawing the Circuit

Placing the Components

We will start by placing two sets of traffic lights and a PIC16F84 on a new schematic layout. Start a fresh design, select the Main Mode and the Component icons (all the icons have both tooltip text and context-sensitive help to assist in their use) and then left click on the letter 'P' above the *Object Selector*. The *Device Library* selector will now appear over the *Editing Window* (for more information see *Picking, Placing and Wiring Up Components* in the ISIS manual).

The Traffic Lights can be found in the ACTIVE Library and the PIC microcontroller in the MICRO Library. To select a component into the design, highlight the component name in the *Objects* listbox and double click on it. A successful selection will result in the component name appearing in the *Object Selector*.

Once you have selected both TRAFFIC LIGHTS and PIC16F84 into the design close the *Device Library Selector* and click left once on the PIC16F84 in the *Object Selector* (this should highlight your selection and a preview of the component will appear in the *Overview Window* at the top right of the screen). Now left click on the *Editing Window* to place the component on the schematic - repeat the process to place two sets of traffic lights on the schematic.

Movement and Orientation

We now have the building blocks on the schematic but the chances are they are not ideally positioned. To move a component, point the mouse over it and right click (this should highlight the component), then depress the left mouse button and drag (you should see the component outline 'follow' the mouse pointer) to the desired position. When you have the outline where you want release the left mouse button and the component will move to the specified position. Note that at this point the component is still highlighted - right click on any empty area of the *Editing Window* to restore the component to it's normal status.

To orient a component, right click over it in the same way as before then click on one of the *Rotation* icons. This will rotate the component through 90 degrees - repeat as necessary. Again, it is good practice to right click on an empty area of the schematic when you are finished to restore the component to it's original state.

Set out the schematic in a sensible fashion (perhaps based on the sample given), moving and orientating the components as required. If you are having problems we advise you to work through the tutorial in the ISIS manual

Zooming and Snapping

In order to wire up the schematic it is useful to be able to zoom in to a specific area. Hitting the F6 key will zoom around the current position of the mouse, or, alternatively, holding down the SHIFT key and dragging a box with the left mouse button will zoom in on the contents of the dragged area. To zoom back out again hit the F7 key, or, should you wish to zoom out until you can see the entire design, hit the F8 key. Corresponding commands can be accessed through the *View Menu*.

ISIS has a very powerful feature called *Real Time Snap*. When the mouse pointer is positioned near to pin ends or wires, the cursor location will be snapped onto these objects. This allows for easy editing and manipulation of the schematic. This feature can be found in the *Tools Menu* and is enabled by default.

More information on zooming and snapping can be found in *The Editing Window* in the ISIS manual.

Wiring Up

The easiest way to wire a circuit is to use the *Wire Auto Router* option on the *Tools Menu*. Make sure that this is enabled (a tick should be visible to the left of the menu option). More information on the functionality on the WAR can be found in the ISIS manual.

Zoom in to the PIC until all the pins are comfortably visible and then place the mouse pointer over the end of pin 6 (RB0/INT). You should see a small 'x' cursor on the end of the mouse. This indicates that the mouse is at the correct position to connect a wire to this pin. Left click the mouse to start a wire and then move the mouse to the pin connected to the red light on one of the sets of Traffic Lights. When you get an 'x' cursor again over this pin left click to complete the connection. Repeat the process to wire up both sets of Traffic Lights as given on the sample circuit.

There are a couple of points worth mentioning about the wiring up process:

- You can wire up in any mode - ISIS is clever enough to determine what you are doing.
- When enabled, the Wire Autorouter will route around obstacles and generally find a sensible path between connections. This means that, as a general rule, all you need to do is left click at both end points and let ISIS take care of the path between them.
- ISIS will automatically pan the screen is you nudge the edge of the *Editing Window* while placing a wire. This means that you can zoom in to the most suitable level and, so long as you know the approximate position of the target component, simply nudge the screen over until it is in view. Alternatively, you can zoom in and out while placing wires (using the F6 and F7 keys).

Finally, we have to wire pin 4 to a power terminal. Select *Gadgets Mode* and the *Terminal* icon and highlight 'POWER' in the *Object Selector*. Now left click on a suitable spot and place the terminal. Select the appropriate orientation and wire the terminal to pin 4 using the same techniques as before.

More useful information on the wiring up process is available in the tutorial in the ISIS manual.

At this point we recommend that you load the completed version of the circuit – this will avoid any confusion arising if the version you have drawn is different from ours!

Writing the Program

Source Listing

For the purposes of our tutorial, we have prepared the following program which will enable the PIC to control the traffic lights. This program is provided in a file called TL.ASM and can be found in the "Samples\Tutorials" directory.

```
LIST      p=16F84 ; PIC16F844 is the target processor

           #include "P16F84.INC" ; Include header file

           CBLOCK 0x10                ; Temporary storage
               state
               11,12
           ENDC

           org      0                  ; Start up vector.
           goto     setports           ; Go to start up code.

           org      4                  ; Interrupt vector.
halt       goto     halt               ; Sit in endless loop and do
nothing.

setports   clrw                      ; Zero in to W.
           movwf    PORTA             ; Ensure PORTA is zero before
enabled.
           movwf    PORTB             ; Ensure PORTB is zero before
enabled.
           bsf      STATUS,RP0        ; Select Bank 1
```



```
w2      decfsz 12
        goto   w2
        return
        END
```

There is, in fact, a deliberate mistake in the above code, but more of that later...

Attaching the Source File

The next stage is to attach the program to our design in order that we can successfully simulate its behaviour. We do this through the commands on the *Source Menu*. Go to the *Source Menu* now and select the *Add/Remove Source Files* Command. Click on the *New* button, browse until you reach the “Samples\Tutorials” directory and select the TL.ASM file. Click open and the file should appear in the *Source Code Filename* drop down listbox.

We now need to select the code generation tool for the file. For our purposes the MPASM tool will suffice. This option should be available from the drop down listbox and so left clicking will select it in the usual fashion. (Note that If you are planning to use a new assembler or compiler for the first time, you will need to register it using the *Define Code Generation Tools* command).

Finally, it is necessary to specify which file the processor is to run. In our example this will be tl.hex (the hex file produced from MPASM subsequent to assembling tl.asm). To attach this file to the processor, right click on the schematic part for the PIC and then left click on the part. This will bring up the *Edit Component* dialogue form which contains a field for *Program File*. If it is not already specified as tl.hex either enter the path to the file manually or browse to the location of the file via the “?” button to the right of the field. Once you have specified the hex file to be run hit ok to exit the dialogue form.

We have now attached the source file to the design and specified which *Code Generation Tool* will be used. A more detailed explanation on the *Source Code Control System* is given on page 47.

Debugging the Program

Simulating the Circuit

In order to simulate the circuit point the mouse over the *Play Button* on the animation panel at the bottom right of the screen and click left. The status bar should appear with the time that the animation has been active for. You should also notice that one of the traffic lights is green while the other is red and the logic state of the pins can be seen on the schematic. You will notice, however, that the traffic lights are not changing state. This is due to a deliberate bug we have introduced into the code. At this stage it would be appropriate to debug our program and try to isolate the problem.

Debugging Mode

In order to ensure that we are thorough in the debugging of the circuit we will stop the current simulation. Once you have done that you can start debugging by pressing CTRL+F12. Two popup windows should appear - one holding the current register values and the other holding displaying the source code for the program. Either of these can be activated from the *Debug Menu* along with a host of other informative windows. We also want to

activate the *Watch Window* from which we can monitor the appropriate changes in the state variable. A full explanation of this feature is available in the section entitled *The Watch Window* on page 55.

Concentrating for now on the *Source* popup notice the red arrow at the far left. This, together with the highlighted line indicate the current position of the program counter. To set a breakpoint here hit the ENTER key (the breakpoint will always be set at the highlighted line). If we wanted to clear the breakpoint we could do so by hitting the ENTER key again but we will leave it set just now.

Setting a Breakpoint

Looking at the program it can be seen that it loops back on itself in a repeating cycle. It would therefore be a good idea to set a breakpoint at the beginning of this loop before we start. You can do this by highlighting the line (at address 000E) with the mouse, and then pressing F9. Then press F12 to set the program running. You should now see a message on the *Status Bar* indicating that a digital breakpoint has been reached and the Program Counter (PC) address. This should correspond with the address of the first breakpoint we have set.

A list of the debugging keys are given in the *Debug Menu* but for the most part we will be using F11 to single step through the program. Hit the F11 key now and notice that the red arrow at the left has moved down to the next instruction. What we have actually done is executed the 'clrw' instruction and then stopped. You can verify this by looking at the W register in the *Registers Window* and noticing that it has been cleared to zero.

What we now need to do is determine what we expect to happen on execution of the next instruction and then test to see if it actually does happen. For example, the next instruction should move the contents of the 'w' register in to PORT A i.e. Port A should be cleared. Executing this instruction and checking the *Register Window* verifies that this is indeed the case. Continuing in this vein until you reach our second breakpoint you should notice that both ports have been cleared ready for output (as dictated by the TRISB register) and that the state variable has correctly been set to 0.

As this is a function call we have the option of *Stepping Over* the function (by hitting the F10 key) but for completeness we will step through each instruction. Hitting F11 here will jump us to the first executable line of the getmask function. Stepping forward we see that the move operation was successful and that we 'land' in the correct place for adding an offset of zero onto our lookup table. When we return to the main program therefore, we have the mask that we would expect. Single stepping further and writing the mask to the port we can see the correct result on the schematic. Single stepping again to increment the state is also successful as evidenced by the *Register Window* where the value for the W register is incremented by 1.

The single step takes us onto the instruction designed to wrap the state around to zero when it is incremented above 3. This, as can be seen from the *Watch Window* , is not performing

as it should. The state should clearly be incremented to state 1 here in order for the mask to be set correctly on the next execution of the loop.

Finding the Bug

Closer investigation reveals that the problem is caused by ANDing with 4 instead of with 3. The states we want are 0,1,2, 3 and any of these when ANDed with 4 gives 0. This is why when running the simulation the state of the traffic lights does not change. The solution is simply to change the problem instruction to AND the state with 3 instead of 4. This means that the state will increment to 3 and when the W register is incremented to 4 the state will wrap around to 0. An alternative solution would be to simply test for the case when the 'W' register hits 4 and to reset it to zero.

While this short example of the debugging techniques available in Proteus VSM illustrates the basics there is a lot of additional functionality available. It is recommended that you look at the section on *Source Level Debugging* later in this documentation for a more detailed explanation.

GRAPH BASED SIMULATION TUTORIAL

Introduction

The purpose of this tutorial is to show you, by use of a simple amplifier circuit, how to perform a graph based simulation using PROTEUS VSM. It takes you, in a step-by-step fashion, through:

- Placing graphs, probes and generators.
- Performing the actual simulation.
- Using graphs to display results and take measurements.
- A survey of the some of the analysis types that are available.

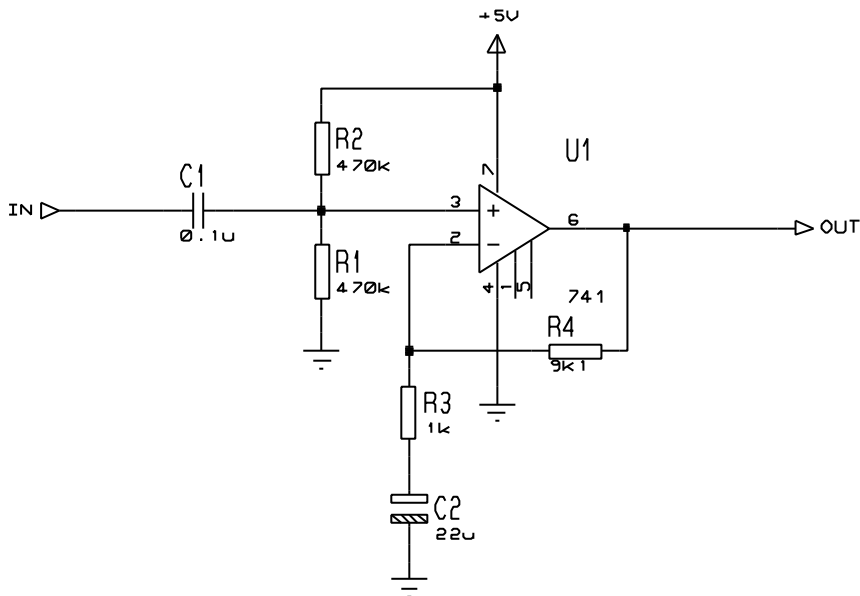
The tutorial does **not** cover the use of ISIS in its general sense - that is, procedures for placing components, wiring them up, tagging objects, etc. This is covered to some extent in the *Interactive Simulation Tutorial* and in much greater detail within the ISIS manual itself. If you have not already made yourself familiar with the use of ISIS then you must do so before attempting this tutorial.

We do strongly urge you to work right the way through this tutorial before you attempt to do your own graph based simulations: gaining a grasp of the concepts will make it much easier to absorb the material in the reference chapters and will save much time and frustration in the long term.

Getting Started

The circuit we are going to simulate is an audio amplifier based on a 741 op-amp, as shown overleaf. It shows the 741 in an unusual configuration, running from a single 5 volt supply. The feedback resistors, R3 and R4, set the gain of the stage to be about 10. The input bias components, R1, R2 and C1, set a false ground reference at the non-inverting input which is decoupled from the signal input.

As is normally the case, we shall perform a *transient analysis* on the circuit. This form of analysis is the most useful, giving a large amount of information about the circuit. Having completed the description of simulation with transient analysis, the other forms of analysis will be contrasted.



If you want, you can draw the circuit yourself, or you can load a ready made design file from "Samples\Tutorials\ASIMTUT1.DSN" within your Proteus installation. Whatever you choose, at this point ensure you have ISIS running and the circuit drawn.

Generators

To test the circuit, we need to provide it with a suitable input. We shall use a voltage source with a square wave output for our test signal. A *generator* object will be used to generate the required signal.

To place a generator, first click left on the *Generator* icon: the *Object Selector* displays a list of the available generator types. For our simulation, we want a *Pulse* generator. Select the *Pulse* type, move the mouse over to the edit window, to the right of the IN terminal, and click left on the wire to place the generator.

Generator objects are like most other objects in ISIS; the same procedures for previewing and orienting the generator before placement and editing, moving, re-orienting or deleting the object after placement apply (see *GENERAL EDITING FACILITIES* in the ISIS manual or *GENERATORS AND PROBES* on page 93 in this manual).

As well as being dropped onto an existing wire, as we just did, generators may be placed on the sheet, and wired up in the normal manner. If you drag a generator off a wire, then ISIS assumes you want to detach it, and does not drag the wire along with it, as it would do for components.

Notice how the generator is automatically assigned a reference - the terminal name IN. Whenever a generator is wired up to an object (or placed directly on an existing wire) it is assigned the name of the *net* to which it is connected. If the net does not have a name, then the name of the nearest component pin is used by default.

Finally, we must edit the generator to define the pulse shape that we want. To edit the generator, tag it with the right mouse button and then click left on it to access its *Edit Generator* dialogue form. Select the *High Voltage* field and set the value to 10mV. Also set the pulse width to 0.5s.

Select the OK button to accept the changes. *GENERATORS AND PROBES* on page 93 gives a complete reference of the properties recognised by all the types of generator. For this circuit only one generator is needed, but there is no limit on the number which may be placed.

Probes

Having defined the input to our circuit using a generator, we must now place probes at the points we wish to monitor. We are obviously interested in the output, and the input after it has been biased is also a useful point to probe. If needs be, more probes can always be added at key points and the simulation repeated.

To place a probe, first click left on the *Voltage Probe* icon (ensure you have not selected a current probe by accident - we shall come to these later). Probes can be placed onto wires, or placed and then wired, in the same manner as generators. Move the mouse over to the edit

window, to the left of U1 pin 3, and click left to place the probe on the wire joining pin 3 to R1 and R2. Be sure to place the probe on the wire, as it cannot be placed on the pin itself. Notice the name it acquires is the name of the nearest device to which it is connected, with the pin name in brackets. Now place the second probe by clicking left just to the left of the OUT terminal, on the wire between the junction dot and the terminal pin.

Probe objects are like generators and most other objects in ISIS; the same procedures for previewing and orienting the probe before placement, and editing, moving, re-orienting or deleting the probe after placement apply (see *GENERAL EDITING FACILITIES* in the ISIS manual or PROBES on page 101 in this manual). Probes may be edited in order to change their reference labels. The names assigned by default are fine in our case, but a useful tip when tagging probes is to aim for the *tip* of the probe, not the body or reference label.

Now that we have set up the circuit ready for simulation, we need to place a graph to display the results on.

Graphs

Graphs play an important part in simulation: they not only act as a display medium for results but actually define what simulations are carried out. By placing one or more graphs and indicating what sort of data you expect to see on the graph (digital, voltage, impedance, etc.) ISIS knows what type or types of simulations to perform and which parts of a circuit need to be included in the simulation. For a transient analysis we need an *Analogue* type graph. It is termed analogue rather than transient in order to distinguish it from the *Digital* graph type, which is used to display results from a digital analysis, which is really a specialised form of transient analysis. Both can be displayed against the same time axis using a *Mixed* graph.

To place a graph, first select the *Graph* icon: the *Object Selector* displays a list of the available graph types. Then select the *Analogue* type, move the mouse over to the edit window, click (and hold down) the left mouse button, drag out a rectangle of the appropriate size, and then release the mouse button to place the graph.

Graphs behave like most objects in ISIS, though they do have a few subtleties. We will cover the features pertinent to the tutorial as they occur, but the reference chapter on graphs is well worth a read. You can tag a graph in the usual way with the right mouse button, and then (using the left mouse button) drag one of the handles, or the graph as a whole, about to resize and/or reposition the graph.

We now need to add our generator and probes on to the graph. Each generator has a probe associated with it, so there is no need to place probes directly on generators to see the input wave forms. There are three ways of adding probes and generators to graphs:

- The first method is to tag each probe/generator in turn and drag it over the graph and release it - exactly as if we were repositioning the object. ISIS detects that you are trying

to place the probe/generator over the graph, restores the probe/generator to its original position, and adds a trace to the graph with the same reference as that of the probe/generator. Traces may be associated with the left or right axes in an analogue graph, and probes/generators will add to the axis nearest the side they were dropped. Regardless of where you drop the probe/generator, the new trace is always added at the bottom of any existing traces.

The second and third method of adding probes/generators to a graph both use the *Add Trace* command on the graph menu; this command always adds probes to the current graph (when there is more than one graph, the current graph is the one currently selected on the *Graph* menu).

- If the *Add Trace* command is invoked without any tagged probes or generators, then the *Add Transient Trace* dialogue form is displayed, and a probe can be selected from a list of all probes in the design (including probes on other sheets).
- If there are tagged probes/generators, invoking the *Add Trace* command causes you to be prompted to *Quick Add* the tagged probes to the current graph; selecting the *No* option invokes the *Add Transient Trace* dialogue form as previously described. Selecting the *Yes* option adds all tagged probes/generators to the current graph in alphabetical order.

We will *Quick Add* our probes and the generator to the graph. Either tag the probes and generators individually, or, more quickly, drag a tag box around the entire circuit - the Quick Add feature will ignore all tagged objects other than probes and generators. Select the *Add Trace* option from the *Graph* menu and answer *Yes* to the prompt. The traces will appear on the graph (because there is only one graph, and it was the last used, it is deemed to be the *current* graph). At the moment, the traces consist of a name (on the left of the axis), and an empty data area (the main body of the graph). If the traces do not appear on the graph, then it is probably too small for ISIS to draw them in. Resize the graph, by tagging it and dragging a corner, to make it sufficiently big.

As it happens, our traces (having been placed in alphabetical order) have appeared in a reasonable order. We can however, shuffle the traces about. To do this, ensure the graph is *not* tagged, and click right over the name of a trace you want to move or edit. The trace is highlighted to show that it is tagged. You can now use the left mouse button to drag the trace up or down or to edit the trace (by clicking left without moving the mouse) and the right button to delete the trace (i.e. remove it from the graph). To untag all traces, click the right mouse button anywhere over the graph, but *not* over a trace label (this would tag or delete the trace).

There is one final piece of setting-up to be done before we start the simulation, and this is to set the simulation run time. ISIS will simulate the circuit according to the end time on the x-scale of the graph, and for a new graph, this defaults to one second. For our purposes, we

want the input square wave to be of fairly high audio frequency, say about 10kHz. This needs a total period of 100 μ s. Tag the graph and click left on it to bring up its *Edit Transient Graph* dialogue form. This form has fields that allow you to title the graph, specify its simulation start and stop times (these correspond to the left and right most values of the x axis), label the left and right axes (these are not displayed on *Digital* graphs) and also specifies general properties for the simulation run. All we need to change is the stop time from 1.00 down to 100 μ (you can literally type in 100 μ - ISIS will covert this to 100E-6) and select OK.

The design is now ready for simulation. At this point, it is probably worthwhile loading our version of the design ("Samples\Tutorials\ASIMTUT2.DSN") so as to avoid any problems during the actual simulation and subsequent sections. Alternatively, you may wish to continue with the circuit you have entered yourself, and only load the ASIMTUT2.DSN file if problems arise.

Simulation

To simulate the circuit, all you need do is invoke the *Simulate* command on the *Graph* menu (or use its keyboard short-cut: the space bar). The *Simulate* command causes the circuit to be simulated and the *current* graph (the one marked on the *Graph* menu) to be updated with the simulation results.

Do this now. The status bar indicates how far the simulation process has reached. When the simulation is complete, the graph is redrawn with the new data. For the current release of ISIS and simulator kernels, the start time of a graph is ignored - simulation always starts at time zero and runs until the stop time is reached or until the simulator reaches a quiescent state. You can abort a simulation mid-way through by pressing the ESC key.

A simulation log is maintained for the last simulation performed. You can view this log using the *View Log* command on the *Graph* menu (or the CTRL+'V' keyboard short-cut). The simulation log of an analogue simulation rarely makes for exciting reading, unless warnings or errors were reported, in which case it is where you will find details of exactly what went wrong. In some cases, however, the simulation log provides useful information that is not easily available from the graph traces.

So the first simulation is complete. Looking at the traces on the graph, its hard to see any detail. To check that the circuit is working as expected, we need to take some measurements...

Taking Measurements

A graph sitting on the schematic, alongside a circuit, is referred to as being *minimised*. To take timing measurements we must first *maximise* the graph. To do this, first ensure the graph is *not* tagged, and then click the left mouse button on the graph's title bar; the graph is redrawn in its own window. Along the top of the display, the menu bar is maintained. Below this, on the left side of the screen is an area in which the trace labels are displayed and to right of this are the traces themselves. At the bottom of the display, on the left is a toolbar, and to the right of this is a status area that displays cursor time/state information. As this is a new graph, and we have not yet taken any measurements, there are no cursors visible on the graph, and the status bar simply displays a title message.

The traces are colour coded, to match their respective labels. The OUT and U1(POS IP) traces are clustered at the top of the display, whilst the IN trace lies along the bottom. To see the traces in more detail, we need to separate the IN trace from the other two. This can be achieved by using the left mouse button to drag the trace *label* to the right-hand side of the screen. This causes the right y-axis to appear, which is scaled separately from the left. The IN trace now seems much larger, but this is because ISIS has chosen a finer scaling for the right axis than the left. To clarify the graph, it is perhaps best to remove the IN trace altogether, as the U1(POS IP) is just as useful. Click right on the IN label twice to delete it. The graph now reverts to a single, left hand side, y-axis.

We shall measure two quantities:

- The voltage gain of the circuit.
- The approximate fall time of the output.

These measurements are taken using the *Cursors*.

Each graph has two cursors, referred to as the *Reference* and *Primary* cursors. The reference cursor is displayed in red, and the primary in green. A cursor is always 'locked' to a trace, the trace a cursor is locked to being indicated by a small 'X' which 'rides' the waveform. A small mark on both the x- and y-axes will follow the position of the 'X' as it moves in order to facilitate accurate reading of the axes. If moved using the keyboard, a cursor will move to the next small division in the x-axis.

Let us start by placing the *Reference* cursor. The same keys/actions are used to access both the *Reference* and *Primary* cursors. Which is actually affected is selected by use of the CTRL key on the keyboard; the *Reference* cursor, as it is the least used of the two, is always accessed with the CTRL key (on the keyboard) pressed down. To place a cursor, all you need to do is point at the trace data (*not* the trace label - this is used for another purpose) you want to lock the cursor to, and click left. If the CTRL key is down, you will place (or move) the *Reference* cursor; if the CTRL key is not down, then you will place (or move) the *Primary*

cursor. Whilst the mouse button (and the CTRL key for the *Reference* cursor) is held down, you can drag the cursor about. So, hold down (and keep down) the CTRL key, move the mouse pointer to the right hand side of the graph, above both traces, and press the left mouse button. The red *Reference* cursor appears. Drag the cursor (still with the CTRL key down) to about 70u or 80u on the x-axis. The title on the status bar is removed, and will now display the cursor time (in red, at the left) and the cursor voltage along with the name of the trace in question (at the right). It is the OUT trace that we want.

You can move a cursor in the X direction using the left and right cursor keys, and you can lock a cursor to the previous or next trace using the up and down cursor keys. The LEFT and RIGHT keys move the cursor to the left or right limits of the x-axis respectively. With the control key still down, try pressing the left and right arrow keys on the keyboard to move the *Reference* cursor along small divisions on the time axis.

Now place the *Primary* cursor on the OUT trace between 20u and 30u. The procedure is exactly the same as for the *Reference* cursor, above, except that you do not need to hold the CTRL key down. The time and the voltage (in green) for the primary cursor are now added to the status bar.

Also displayed are the differences in both time and voltage between the positions of the two cursors. The voltage difference should be a fraction above 100mV. The input pulse was 10mV high, so the amplifier has a voltage gain of 10. Note that the value is positive because the *Primary* cursor is above the *Reference* cursor - in delta read-outs the value is *Primary* minus *Reference*.

We can also measure the fall time using the relative time value by positioning the cursors either side of the falling edge of the output pulse. This may be done either by dragging with the mouse, or by using the cursor keys (don't forget the CTRL key for the *Reference* cursor). The *Primary* cursor should be just to the right of the curve, as it straightens out, and the *Reference* cursor should be at the corner at the start of the falling edge. You should find that the falling edge is a little under 10μs.

Using Current Probes

Now that we have finished with our measurements, we can return to the circuit - just close the graph window in the usual way, or for speed you can press ESC on the keyboard. We shall now use a current probe to examine the current flow around the feedback path, by measuring the current into R4.

Current probes are used in a similar manner to voltage probes, but with one important difference. A current probe needs to have a *direction* associated with it. Current probes work by effectively breaking a wire, and inserting themselves in the gap, so they need to know which way round to go. This is done simply by the way they are placed. In the default orientation (leaning to the right) a current probe measures current flow in a *horizontal* wire

from left to right. To measure current flow in a vertical wire, the probe needs to be rotated through 90° or 270°. Placing a probe at the wrong angle is an error, that will be reported when the simulation is executed. If in doubt, look at the arrow in the symbol. This points in the direction of current flow.

Select a current probe by clicking on the *Current Probe* icon. Click left on the clockwise *Rotation* icon such that the arrow points downwards. Then place the probe on the vertical wire between the right hand side of R4 and U1 pin 6. Add the probe to the right hand side of the graph by tagging and dragging onto the right hand side of the minimised graph. The right side is a good choice for current probes because they are normally on a scale several orders of magnitude different than the voltage probes, so a separate axis is needed to display them in detail. At the moment no trace is drawn for the current probe. Press the space bar to re-simulate the graph, and the trace appears.

Even from the minimised graph, we can see that the current in the feedback loop follows closely the wave form at the output, as you would expect for an op-amp. The current changes between 10µA and 0 at the high and low parts of the trace respectively. If you wish, the graph may be maximised to examine the trace more closely.

Frequency Analysis

As well as transient analysis, there are several other analysis types available in analogue circuit simulation. They are all used in much the same way, with graphs, probes and generators, but they are all different variations on this theme. The next type of analysis that we shall consider is *Frequency* analysis. In frequency analysis, the x-axis becomes frequency (on a logarithmic scale), and both magnitude and phase of probed points may be displayed on the y-axes.

To perform a frequency analysis a *Frequency* graph is required. Click left on the *Graph* icon, to re-display the list of graph types in the object selector, and click on the *Frequency* graph type. Then place a graph on the schematic as before, dragging a box with the left mouse button. There is no need to remove the existing transient graph, but you may wish to do so in order to create some more space (click right twice to delete a graph).

Now to add the probes. We shall add both the voltage probes, OUT and U1(POS IP). In a frequency graph, the two y-axes (left and right) have special meanings. The left y-axis is used to display the *magnitude* of the probed signal, and the right y-axis the *phase*. In order to see both, we must add the probes to both sides of the graph. Tag and drag the OUT probe onto the left of the graph, then drag it onto the right. Each trace has a separate colour as normal, but they both have the same name. Now tag and drag the U1(POS IP) probe onto the left side of the graph only.

Magnitude and phase values must both be specified with respect to some reference quantity. In ISIS this is done by specifying a *Reference Generator*. A reference generator always has

an output of 0dB (1 volt) at 0°. Any existing generator may be specified as the reference generator. All the other generators in the circuit are ignored in a frequency analysis. To specify the IN generator as the reference in our circuit, simply tag and drag it onto the graph, as if you were adding it as a probe. ISIS assumes that, because it is a generator, you are adding it as the reference generator, and prints a message on status line confirming this. Make sure you have done this, or the simulation will not work correctly.

There is no need to edit the graph properties, as the frequency range chosen by default is fine for our purposes. However, if you do so (by pointing at the graph and pressing CTRL-E), you will see that the *Edit Frequency Graph* dialogue form is slightly different from the transient case. There is no need to label the axes, as their purpose is fixed, and there is a check box which enables the display of magnitude plots in dB or normal units. This option is best left set to dB, as the absolute values displayed otherwise will not be the actual values present in the circuit.

Now press the space bar (with the mouse over the frequency graph) to start the simulation. When it has finished, click left on the graph title bar to maximise it. Considering first the OUT magnitude trace, we can see the pass-band gain is just over 20dB (as expected), and the useable frequency range is about 50Hz to 20kHz. The cursors work in exactly the same manner as before - you may like to use the cursors to verify the above statement. The OUT phase trace shows the expected phase distortion at the extremes of the response, dropping to -90° just off the right of the graph, at the unity gain frequency. The high-pass filter effect of the input bias circuitry can be clearly seen if the U1(POS IP) magnitude trace is examined. Notice that the x-axis scale is logarithmic, and to read values from the axis it is best to use the cursors.

Swept Variable Analysis

It is possible with ISIS to see how the circuit is affected by changing some of the circuit parameters. There are two analysis types that enable you to do this - the *DC Sweep* and the *AC Sweep*. A *DC Sweep* graph displays a series of operating point values against the swept variable, while an *AC Sweep* graph displays a series of single point frequency analysis values, in magnitude and phase form like the *Frequency* graph.

As these forms of analysis are similar, we shall consider just one - a *DC Sweep*. The input bias resistors, R1 and R2, are affected by the small current that flows into U1. To see how altering the value of both of these resistors affects the bias point, a *DC Sweep* is used.

To begin with place a *DC Sweep* graph on an unused space on the schematic. Then tag the U1(POS IP) probe and drag it onto the left of the graph. We need to set the sweep value, and this is done by editing the graph (point at it and press CTRL-E). The *Edit DC Sweep Graph* dialogue form includes fields to set the swept variable name, its start and ending values, and the number of steps taken in the sweep. We want to sweep the resistor values across a range

of say $100\text{k}\Omega$ to $5\text{M}\Omega$, so set the *Start* field to 100k and the *Stop* field to 5M . Click on *OK* to accept the changes.

Of course, the resistors R1 and R2 need to be altered to make them swept, rather than the fixed values they already are. To do this, click right and then left on R1 to edit it, and alter the *Value* field from 470k to X. Note that the swept variable in the graph dialogue form was left at X as well. Click on *OK*, and repeat the editing on R2 to set its value to X.

Now you can simulate the graph by pointing at it and pressing the space-bar. Then, by maximising the graph, you can see that the bias level reduces as the resistance of the bias chain increases. By $5\text{M}\Omega$ it is significantly altered. Of course, altering these resistors will also have an effect on the frequency response. We could have done an *AC Sweep* analysis at say 50Hz in order to see the effect on low frequencies.

Noise Analysis

The final form of analysis available is *Noise* analysis. In this form of analysis the simulator will consider the amount of thermal noise that each component will generate. All these noise contributions are then summed (having been squared) at each probed point in the circuit. The results are plotted against the noise bandwidth.

There are some important peculiarities to noise analysis:

- The simulation time is directly proportional to the number of voltage probes (and generators) in the circuit, since each one will be considered.
- Current probes have no meaning in noise analysis, and are ignored.
- A great deal of information is presented in the simulation log file.
- PROSPICE computes both input and output noise. To do the former, an input reference must be defined - this is done by dragging a generator onto the graph, as with a frequency reference. The input noise plot then shows the equivalent noise at the input for each output point probed.

To perform a noise analysis on our circuit, we must first restore R1 and R2 back to $470\text{k}\Omega$. Do this now. Then select a *Noise* graph type, and place a new graph on an unused area of the schematic. It is really only output noise we are interested in, so tag the OUT voltage probe and drag it onto the graph. As before, the default values for the simulation are fine for our needs, but you need to set the input reference to the input generator IN. The *Edit Noise Graph* dialogue form has the check box for displaying the results in dBs. If you use this option, then be aware that 0dB is considered to be 1 volt r.m.s. Click on *Cancel* to close the dialogue form.

Simulate the graph as before. When the graph is maximised, you can see that the values that result from this form of analysis are typically extremely small (pV in our case) as you might

expect from a noise analysis of this type. But how do you track down sources of noise in your circuit? The answer lies in the simulation log. View the simulation log now, by pressing CTRL+V. Use the down arrow icon to move down past the operating point printout, and you should see a line of text that starts

Total Noise Contributions at ...

This lists the individual noise contributions (across the entire frequency range) of each circuit element that produces noise. Most of the elements are in fact inside the op-amp, and are prefixed with U1_. If you select the *Log Spectral Contributions* option on the *Edit Noise Graph* dialogue form, then you will get even more log data, showing the contribution of each component at each spot frequency.

INTERACTIVE SIMULATION

BASIC SKILLS

The Animation Control Panel



Interactive simulations are controlled from a simple VCR like panel that behaves just like a normal remote control. This control is situated at the bottom right of the screen. If it is not visible you need to select the *Circuit Animation* option from the *Graph* menu. There are four buttons that you use to control the flow of the circuits.

- The PLAY button is used to start the simulator.
- The STEP button allows you to step through the animation at a defined rate. If the button is pressed and released then the simulation advances by one time step; if the button is held down then the animation advances continuously until the button is released. The single step time increment may be adjusted from the *Animated Circuit Configuration* dialog box. The step time capability is useful for monitoring the circuits more closely and seeing in slow motion what is affecting what.
- The PAUSE button suspends the animation which can then be resumed either by clicking the PAUSE button again, or single stepped by pressing the STEP button. The simulator will also enter the paused state if an breakpoint is encountered.

The simulation can also be paused by pressing the PAUSE key on the computer keyboard.

- The STOP button tells PROSPICE to stop doing a real time simulation. All animation is stopped and the simulator is unloaded from memory. All the indicators are reset to their inactive states but the actuators (switches etc.) retain their existing settings.

The simulation can also be stopped by pressing SHIFT-BREAK on the computer keyboard.

During an animation, the current simulation time and average CPU loading are displayed on the status bar. If there is insufficient CPU power to run the simulation in real time, the display will read 100% and the simulation time will cease to advance in real time. Aside from this, no harm results from simulating very fast circuits as the system automatically regulates the amount of simulation performed per animation frame.

Indicators and Actuators

Aside from ordinary electronic components, interactive simulations generally make use of special *Active Components*. These components have a number of graphical states and come in two flavours: *Indicators* and *Actuators*. Indicators display a graphical state which changes according to some measured parameter in the circuit, whilst Actuators allow their state to be determined by the user and then modify some characteristic of the circuit.

Actuators are designated by the presence of small red marker symbols which can be clicked with the mouse to operate the control. If you have a mouse with a wheel, you can also operate actuators by pointing at them and rolling the wheel in the appropriate direction.

Setting Up an Interactive Simulation

Most of the skills required to set up and run an interactive simulation centre around drawing a circuit in ISIS. This subject is best grasped by working through the tutorial in the ISIS manual. However, the process may be summarized as follows:

- Pick the components you want to use from the device libraries using the 'P' button on the *Device Selector*. All the active components (actuators and indicators) are contained in the library ACTIVE.LIB, but you can use any components which have a simulator model of some kind.
- Place the components on the schematic.
- Edit them – click right then click left, or press CTRL-E – in order to assign appropriate values and properties. Many models provided context sensitive help so that information about individual properties can be viewed by placing the caret in the field and pressing F1.
- Micro-processor source code can be brought under the control of PROTEUS VSM using the commands on the *Source* menu. Don't forget also to assign the object code (HEX file) to the micro-processor component on the schematic.
- Wire the circuit up by clicking on the pins.
- Delete components by clicking right twice.
- Move components by clicking right then dragging with the left button.
- Click the PLAY button on the *Animation Control Panel* to run the simulation.

Where you have used virtual instruments, or microprocessor models, popup windows related to this components may be displayed using the commands on the *Debug* menu.

ANIMATION EFFECTS

Overview

As well as any active components in the circuit, a number of other animation effects may be enabled to help you study the circuit operation. These options may be enabled using the *Set Animation Options* command on the *System* menu. The settings are saved with the design.

Pin Logic States

This option displays a coloured square by each pin that is connected to a digital or mixed net. By default, the square is blue for logic 0, red for logic 1 and grey for floating. The colours may be changed via the *Set Design Defaults* command on the *Template* menu.

Enabling this option incurs a modest extra burden on the simulator, but can be very, very useful when used in condition with breakpoints and single stepping as it enables you to study the state of a micro-controllers output port pins as the code is single stepped.

Show Wire Voltage as Colour

This option causes any wire that is part of an analogue or mixed net to be displayed in a colour that represents its voltage. By default, the spectrum ranges from blue at $-6V$ through green at $0V$ to red at $+6V$. The voltages may be changed on the *Set Animation Options* dialogue, and the colours via the *Set Design Defaults* command on the *Template* menu.

Enabling this option incurs a modest extra burden on the simulator but is very effective for helping novices understand a circuits operation, especially when used in conjunction with the *Show Wire Current as Arrows* option.

Show Wire Current as Arrows

This option causes an arrow to be drawn on any wire that carrying current. The direction of the arrow reflects conventional current flow, and it is displayed if the magnitude of the current exceeds a threshold. The default threshold is $1\mu A$, although this can be changed on the *Set Animation Options* dialogue.

Computing the wire currents involves the insertion of a zero value voltage source in every wire segment in the schematic (apart from inside models) and this can create a large number of extra nodes. Consequently the burden on the simulator can be significant. However, the arrows are most useful in teaching basic electricity and electronics where the circuits tend to be fairly simple.

ANIMATION TIMESTEP CONTROL

Overview

Two parameters control how an interactive simulation evolves in real time. The *Animation Frame Rate* determines the number of times that the screen is refreshed per second, whilst the *Animation Timestep* determines by how much simulation is progressed during each frame. For real time operation, the timestep should be set to the reciprocal of the frame rate.

Frames Per Second

Normally, it is unnecessary to adjust the frame rate as the default value of 20 frames per second gives smooth animation without overburdening the (considerable) graphics performance of a modern PC. It is, however, occasionally useful to slow it right down when debugging the operation of animated compute models.

Timestep Per Frame

The *Animation Timestep* can be used to make fast circuits run slowly, or slow circuits run quickly. For real time operation, the timestep should be set to the reciprocal of the frame rate.

Of course, the entered timestep will always be a desired value. Achieving it is dependent on there being sufficient CPU power to compute the necessary simulation events within the time allocated to each frame. The CPU load figure displayed during animation represents the ratio of these two times. If insufficient CPU power is available, the CPU load will be 100%, and the *achieved* timestep per frame will drop off.

Single Step Time

The remaining timestep control parameter is the *Single Step Time*. This is the time by which the simulation will advance when the single step button on the *Animation Control Panel* is pressed.

HINTS AND TIPS

Circuit Timescale

Interactive simulations will be generally be viewed in real time, so it is no use drawing circuits with 1MHz clocks, or 10kHz sine wave inputs unless you also adjust the *Timestep per Frame* value in the *Animated Circuits Configuration* dialog box.

If you do attempt to simulate something that operates very quickly, there are several things to bear in mind:

- Given finite CPU power, only a certain amount of simulation time can be computed in a fixed amount of real time. PROTEUS VSM is designed to maintain its animation frame rate (frames per second) whatever, and to cut short any frames which are not completed in the available time. The consequence of this is that very fast circuits will simulate slowly (relative to real time) but smoothly.
- Analogue component models simulate very much more slowly than digital ones. On a fast PC you can simulate digital circuitry operating up to several MHz in real time but analogue circuit electronics will manage only about 10-20kHz.

Consequently it is very silly to attempt to simulate clock oscillators for digital circuitry in the analogue domain. Instead, use a *Digital Clock* generate and set its *Isolate Before* checkbox so that the analogue clock is not simulated.

Voltage Scaling

If you intend to use coloured wires to indicate node voltages you need to give some thought as to the range of voltages that will occur in your circuit. The default range for displayed voltages is $\pm 6V$ so if the circuit operates at significantly different voltages from this you may need to change the *Maximum Voltage* value in the *Animated Circuits Configuration* dialog box.

Earthing

PROSPICE will attempt to define a sensible earth point for any active circuit which does not specifically include a ground terminal. In practice this is usually chosen as the mid-point of the battery, or the centre tap if the circuit has a split supply. It follows that the positive terminal of the battery will sit above ground and the negative terminal below it, corresponding to red and blue wire colours. However, if this behaviour is not what is required, you always have the option of explicitly defining the ground reference point using a ground terminal in ISIS.

High Impedance Points

The automatic earthing logic also checks for any device terminals which are not connected to ground, and will insert high value resistors automatically in order to ensure convergence of the SPICE simulation. This means that ill formed circuits (e.g. circuits with unconnected, or partially connected) components will generally simulate – although there can sometimes be strange results.

VIRTUAL INSTRUMENTS

VOLTMETERS & AMMETERS

A number of interactive voltmeter and ammeter models are provided in the ACTIVE device library. These operate in real time and can be wired into the circuit just like any other component. Once the simulation is started they display the voltage across their terminals or the current flowing through them in an easy to read digital format.

The supplied models cover FSDs of 100, 100m and 100u with a resolution of 3 significant figures and a maximum number of two decimal places. Thus the VOLTMETER object can display values from 0.01V up to 99.9V, whilst the AMMETER-MILLI can display from 0.01mA to 100mA and so on.

The voltmeter models support an internal load resistance property which defaults to 100M but can be changed by editing the component in the usual way. Leaving the value blank disables the load resistance element of the model.

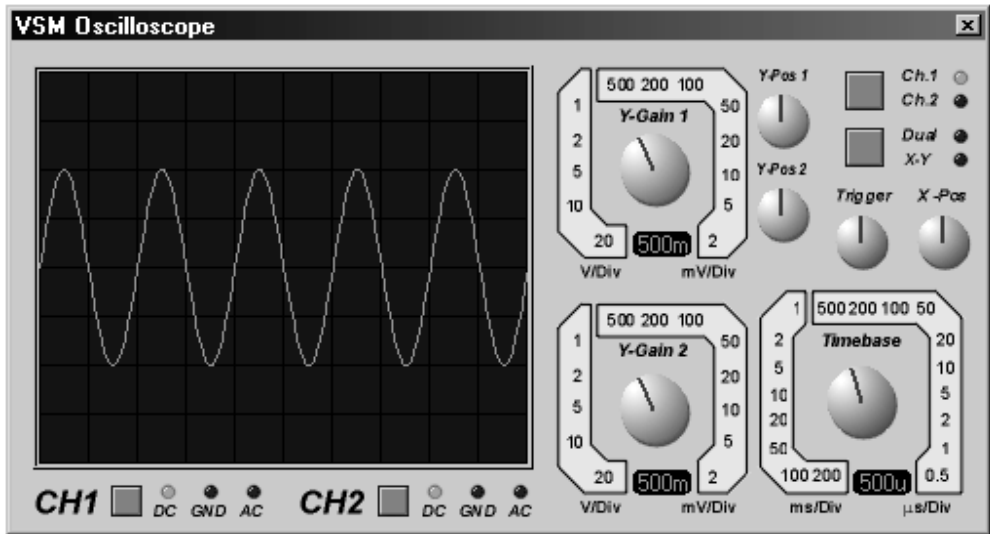
The AC voltmeters and ammeters display true RMS values integrated over a user definable time constant.

OSCILLOSCOPE

Overview

The VSM Oscilloscope is supplied as standard with all versions of ProSPICE and models a basic dual beam analogue unit and is specified as follows:


- Dual channel or X-Y operation.
- Channel gain from 20V/div to 2mV/div
- Timebase from 200ms/div to 0.5us/div
- Automatic voltage level triggering locked to either channel.
- AC or DC coupled inputs.



Using the Oscilloscope

To display analogue waveforms:

1. Pick the OSCILLOSCOPE object from the ACTIVE component library, place one on the schematic and wire its inputs to the signals you want to record.
2. Start an interactive simulation by pressing the play button on the *Animation Control Panel*. The oscilloscope window should appear.
3. If displaying two signals, select *Dual* mode.
4. Set the *timebase* dial to a suitable value for the circuit. You will need to think about the waveform frequencies present in your circuit, and convert these to cycle times by taking their reciprocal.
5. If displaying signals with a DC offset, select AC mode on either or both input channels as appropriate.
6. Adjust the *Y-gain* and *Y-pos* dials so that the waveforms are of a suitable size and position. If a waveform consists of a small AC signal on top of a large DC voltage you may need to connect a capacitor between the test point and the oscilloscope, as the *Y-pos* controls can compensate for only a certain amount of DC.

7. Decide which channel you want to trigger off, and ensure that the *Ch1* or *Ch2* led is lit, as appropriate.
 8. Rotate the *trigger* dial until the display locks to the required part of the input waveform. It will lock to rising slopes if the dial points up, and falling slopes if it points down.
-  See *Rotary Dials* on page 45 for details of how to adjust the rotary controls featured in the oscilloscope.

Modes of Operation

The oscilloscope can operate in 3 modes, indicated as follows:

- Single Beam - neither *Dual* nor *X-Y* leds are lit. In this mode, the *Ch1* and *Ch2* leds indicate which channel is being displayed.
- Dual Beam – the *Dual* led is lit. In this mode, the *Ch1* and *Ch2* leds indicate which channel is being used for triggering.
- X-Y mode – the *X-Y* led is lit.

The current mode may be cycled through these options by clicking the button next to the *Dual* and *X-Y* mode leds.

Triggering

The VSM oscilloscope provides an automatic triggering mechanism which enables it synchronize the timebase to the incoming waveform.

- Which input channel is used for triggering is indicated by the *Ch1* and *Ch2* leds.
- The *trigger* dial rotates continuously round 360 degrees and sets the voltage level and slope at which triggering occurs. When the mark is pointing upwards, the scope triggers on rising voltages; when the mark points downwards it triggers on a falling slope.
- If no triggering occurs for more than 1 timebase period, the timebase will free run.

Input Coupling

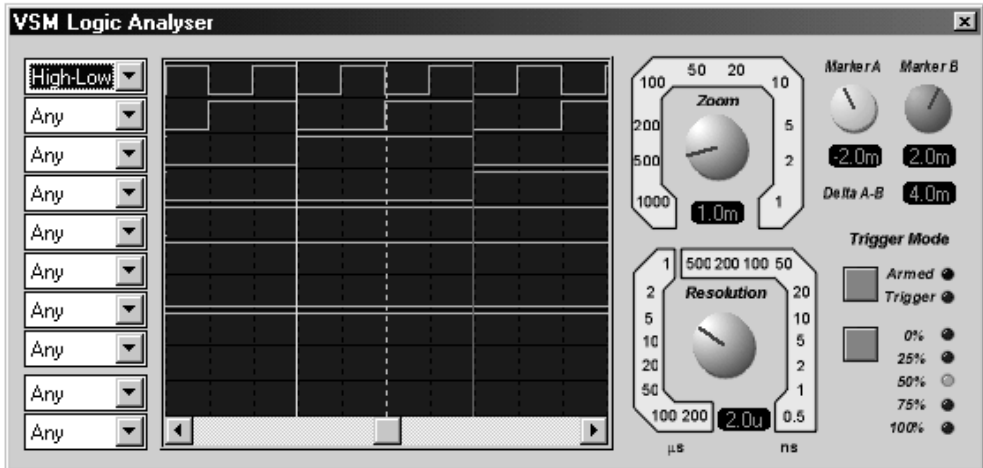
Each input channel can either be directly coupled (DC coupling) or coupled through a simulated capacitance (AC coupling). The latter mode is useful for displaying signals which carry a small AC signal on top of a much larger DC bias voltage.

The inputs may also be temporarily grounded which can be useful when aligning them to the graticule prior to taking measurements.

LOGIC ANALYSER

Overview

The VSM Logic Analyser is supplied as standard with PROTEUS VSM Professional, but is an optional extra for PROTEUS VSM Lite.



A logic analyser operates by continuously recording incoming digital data into a large *capture buffer*. This is a sampling process, so there is an adjustable *resolution* which defines the shortest pulse that can be recorded. A *triggering section* monitors the incoming data and causes the data capture process to stop a certain time after the triggering condition has arisen.; capturing is started by *arming* the instrument. The result is that the contents of the capture buffer both before and after the trigger time can be displayed. Since the capture buffer is very large (10000 samples, in this case) a means of zooming and panning the display is provided. Finally, movable *measurement markers* allow accurate timing measurements of pulse widths and so forth.

The VSM Logic Analyser models a basic 24 channel unit specified as follows:

- 8 x 1 bit traces and 2 x 8 bit bus traces.
- 10000 x 24 bit capture buffer.
- Capture resolution from 200us per sample to 0.5ns per sample with corresponding capture times from 2s to 5ms.
- Display zoom range from 1000 samples per division in to 1 sample per division.

- Triggering on ANDed combination of input states and/or edges, and bus values.
- Trigger positions at 0, 25, 50, 75 and 100% of the capture buffer.
- Two cursors provided for taking accurate timing measurements.

Using the Logic Analyser

To capture and display digital data:

1. Pick the LOGIC ANALYSER object from the ACTIVE component library, place one on the schematic and wire its inputs to the signals you want to record.
2. Start an interactive simulation by pressing the play button on the *Animation Control Panel*. The logic analyser popup window should appear.
3. Set the *resolution* dial to a value suitable for your application. This represents the smallest width of pulse that can be recorded. The finer the resolution, the shorter will be the time during which data is captured.
4. Set the combo-boxes on the left of the instrument to define the required *trigger condition*. For example, if you want to trigger the instrument when the signal connected to channel 1 is high and the signal connected to channel 3 is a rising edge, you would set the first combo-box to “High” and the third combo-box to “Low-High”.
5. Decide whether you want to view data mainly before or after the trigger condition occurs, and click the button next to the *percentage* led to selected the required trigger position.
6. When you are ready, click the button to the left of the *armed* led to arm the instrument.

The *armed* led will light and the *trigger* led will be extinguished. The logic analyser will now capture incoming data continuously whilst monitoring the inputs for the trigger condition. When this occurs, the *trigger* led will light. Data capture will then continue until the part of the capture buffer after the trigger position is full, at which point the *armed* led will extinguish and the captured data will appear in the display.

Panning and Zooming

Since the capture buffer holds 10000 samples, and the display is only 250 pixels wide, a means is needed to pan and zoom within the capture buffer. The *Zoom* dial determines the number of samples per division, whilst the scroll-bar attached to the display itself allows you to pan left and right.

Note that the readout located beneath the *Zoom* dial displays the current time per division in seconds, not the actual setting of the dial itself. The division time is calculated by multiplying the zoom setting by the resolution.

Taking Measurements

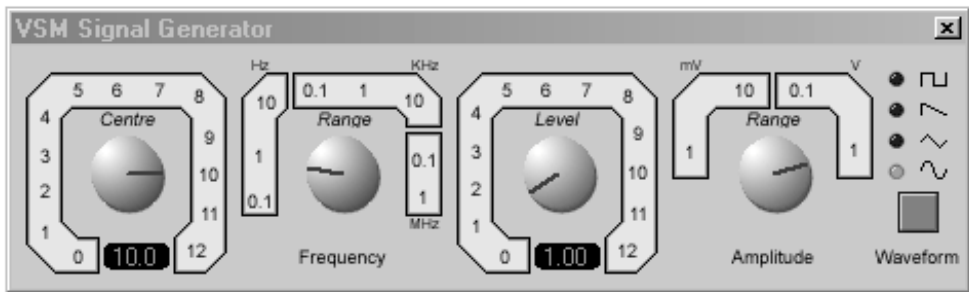
Two adjustable markers are provided which can be used for taking accurate time measurements. Each marker can be positioned using the correspondingly coloured dial. The readout below each dial displays the time point occupied by the marker relative to the trigger time, whilst the *Delta A-B* readout displays the time difference between the markers.

See *Rotary Dials* on page 45 for details of how to adjust the rotary controls featured in the logic analyser.

SIGNAL GENERATOR

Overview

The VSM Signal Generator is supplied as standard with both ProSPICE Professional and ProSPICE Lite.



The VSM signal generator models a simple audio functional generator with the following features:

- Square, saw-tooth, triangle and sine output waveforms.
- Output frequency range from 0-12MHz in 8 ranges
- Output amplitude from 0-12V in 4 ranges.
- Amplitude and frequency modulation inputs.

Using the Signal Generator

To set up a simple audio signal source

1. Pick the SIGNAL GENERATOR object from the ACTIVE component library, place one on the schematic and wire its outputs into the rest of the circuit. In most circumstances (i.e. unless the circuit you are driving requires a balance input source), you will want to

ground the -ve terminal of the generator. You most easily achieve this using a ground terminal.

The amplitude and frequency modulation inputs can be left unconnected unless you are using these features.

3. Start an interactive simulation by pressing the play button on the *Animation Control Panel*. The signal generator popup window should appear.
3. Set the *frequency range* dial to a value suitable for your application. The range values indicate the frequency that is generated when the *centre* vernier control is set at 1.
4. Set the *amplitude range* dial to a value suitable for your application. The range values indicate the amplitude that is generated when the *level* vernier control is set at 1. The amplitude values represent peak output levels.
5. Push the *waveform* button until the LED next to the appropriate waveform icon is lit.

Using the AM & FM Modulation Inputs

The signal generator model supports both amplitude and frequency modulation of the output waveform. Both the amplitude and frequency inputs have the following features:

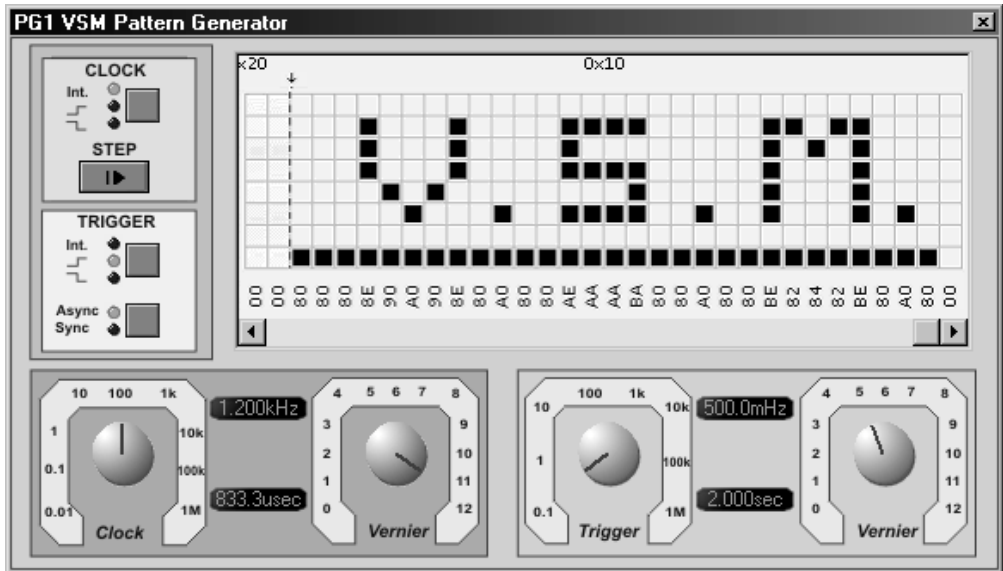
- The gain of the modulation input in terms of Hz/V or V/V is set by the *Frequency Range* and *Amplitude Range* controls respectively.
- The modulation input voltages are clamped at +/- 12V.
- The modulation inputs have infinite input impedance.
- The voltage at the modulation input is added to the setting of the appropriate vernier control before being multiplied by the range setting to determine the instantaneous output frequency of amplitude.

For example, if the frequency range is set at 1KHz, and the frequency vernier is set at 2.0, then a frequency modulation level of 2V will give an output frequency of 4kHz.

DIGITAL PATTERN GENERATOR

Overview

The VSM Pattern Generator is the digital equivalent of the analogue Signal Generator and is provided as standard with all professional versions of the Proteus simulation software.






The VSM Pattern Generator allows for an 8-bit pattern of up to 1K bytes and supports the following main features:

- Will run in either graph based or interactive mode.
- Internal and External Clocking and Trigger modes.
- Vernier adjustment for both clock and trigger dials.
- Hexadecimal or Decimal grid display modes.
- Direct entry of specific values for greater accuracy.
- Loading and saving of pattern scripts.
- Manual specification of pattern period length.
- Single Step control allows you to advance the pattern incrementally.

- Tooltip Display allows you to see exactly where you are on the grid.
- Ability to externally hold the pattern in its current state.
- Block Editing commands on the grid for easier pattern configuration.

Using the Pattern Generator

Outputting a pattern with the Pattern Generator in an interactive simulation.

1. Pick the *PATTGEN* component from the Active Library, place on the schematic and wire to the rest of the circuit.
 2. Initialise an interactive simulation by pressing the pause button on the *Animation Control Panel*. The Pattern Generator window should appear.
 3. Specify the pattern that you want to output by left clicking on grid squares to toggle their logic states.
 4. Decide whether you are clocking internally or externally and set the clock mode button to reflect your choice.
 5. If you are clocking internally adjust the clock dials to specify the desired clock frequency.
 6. Decide whether you are triggering internally or externally and use the trigger buttons to set the mode accordingly. If you are triggering externally you need to think about whether you want to trigger asynchronously or synchronously with the clock.
 7. If you are triggering internally adjust the trigger dials to specify the desired trigger frequency.
 8. Press the play button on the *Animation Control Panel* to output the pattern.
 9. To advance the pattern by a single clock cycle press the suspend button on the *Animation Control Panel* and then the step button to the left of the grid.
-  See *Rotary Dials* on page 45 for details of how to adjust the rotary controls featured in the Pattern Generator.
-  See Trigger Modes, below, for details of the different trigger modes available with the Pattern Generator.
-  See *Additional Functionality*, below, for more information on configuring and using the Pattern Generator.

Outputting a pattern with the Pattern Generator in a graph based simulation.

1. Set up the circuit in the normal way.
2. Insert probes on the schematic at points of interest and add those probes to the graph.
3. Right click and then left click on the Pattern Generator on the schematic to get the *Edit Component* dialogue form.
4. Configure the Trigger and Clock options.
5. Load the desired pattern file into the Pattern Generator Script field.
6. Exit the Pattern Generator Dialogue form and hit the space bar to run the simulation.

The Pattern Generator Component Pins

Data Output Pins (Tri-State Output)

The Pattern Generator can output either on a bus and/or on individual pins.

Clockout Pin (Output)

When the Pattern Generator is clocking internally you can configure this pin to mirror the internal clock pulses. This is set as a property and can be changed via the Edit Component dialogue form. By default this option is disabled as it incurs a performance hit, particularly for high frequency clocks.

Cascade Pin (Output)

The Cascade Pin is driven high when the first bit of the pattern is being driven and remains high until the next bit is driven (one clock cycle later). This means it is high for the first clock cycle on starting the simulation and again for the first clock cycle subsequent to a reset.

Trigger Pin (Input)

This input pin is used to feed an external trigger pulse into the Pattern Generator. There are four external trigger modes which are discussed in detail under *Reset Modes*.

Clock-In Pin (Input)

This input pin is used to apply external timing to the Pattern Generator. There are two external clocking modes which are discussed in greater depth under *Clocking Modes*.

Hold Pin (Input)

This pin, when driven high can be used to pause the Pattern Generator. The pattern will remain at the point it was at until the hold pin is released. For internal clocking and/or trigger

the timing will resume relative to the point at which it was paused. For example, on a 1Hz internal clock, if the pattern is paused at time 3.6 seconds and resumed at 5.2 seconds then the next falling clock edge will be at 5.6 seconds.

Output Enable Pin (Input)

This pin must be driven high in order to enable the output pins. If this pin is not high then the Pattern Generator, while still running the specified pattern, will not drive the pattern onto the output pins.

Clocking Modes

Internal Clocking

Internal clocking is negative edge triggered. That is, the clock pulse will be low-high-low per clock cycle.

Internal clocking can be specified either prior to simulation via the Edit Component dialogue form for the Pattern Generator or during a pause in the simulation via the clock mode button.

The *Clockout* pin, when enabled, mirrors the internal clock. By default, this pin is disabled as it will cause some performance loss (particularly with high frequency clocks) but it can be enabled via the Edit Component dialogue form for the Pattern Generator.

External Clocking

There are two external clocking modes – negative edge (low-high-low) and positive edge (high-low-high).

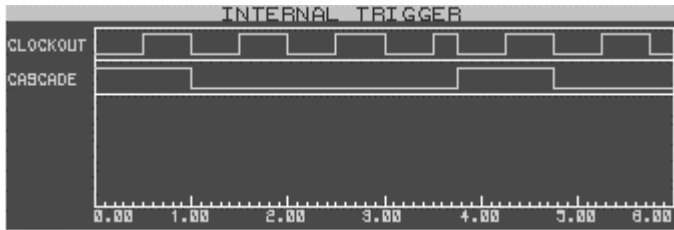
To clock externally wire the clock pulse to the *Clockin* pin and select one of the two external clocking modes.

As with Internal clocking you can change the external clocking mode either by editing the component prior to simulation or via the clock mode button during a pause in the simulation.

Trigger Modes

Internal Trigger

The Internal Trigger Mode in the Pattern Generator triggers the pattern at specified intervals. If the clocking is internal the clock pulse is reset at this point. This behaviour is shown graphically below.

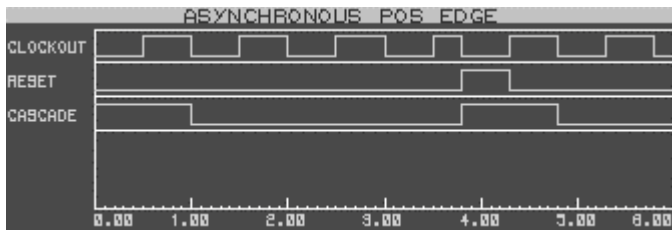


We are clocking internally at 1Hz and the internal trigger time is set to 3.75sec. The Cascade pin is high while the first bit of the pattern is being driven on the pins and low at all other times.

Note that at trigger time the internal clock is asynchronously reset. The first bit of the pattern is driven onto the output pins (as evidenced by the *Cascade Pin* being driven high).

External Asynchronous Positive Edge Trigger

The trigger is specified via a positive edge transition on the *Trigger Pin*. The Trigger actions immediately and the next clock edge will be a low-high transition at time bitclock/2 subsequent to the time of the reset as shown below.



We are clocking internally at 1Hz and the trigger pin goes high at time 3.75sec. Immediately on the positive edge of the trigger pin the clock is reset and the first bit of the pattern is driven onto the pins.

External Synchronous Positive Edge Trigger

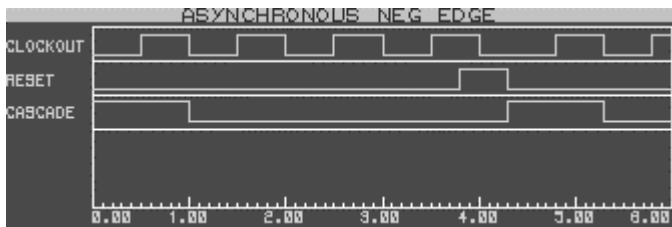
The trigger is specified via a positive edge transition on the *Trigger Pin*. The trigger is latched and will action synchronously with the next falling clock edge as shown below.



We are clocking internally at 1Hz. Note that the clock is unaffected by the trigger and that the trigger actions on the falling clock edge subsequent to the positive edge pulse.

External Asynchronous Negative Edge Trigger

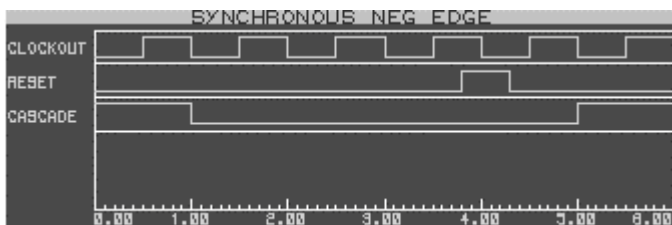
The trigger is specified via a negative edge transition on the *Trigger Pin*. The trigger actions immediately and the first bit of the pattern is driven onto the output pins.



We are clocking internally at 1Hz. We can see that clock resets on the negative edge of the trigger pulse and that the first bit of the pattern is driven at that time.

External Synchronous Negative Edge Trigger

The trigger is specified via a negative edge transition on the *Trigger Pin*. The trigger is latched and actions synchronously with the next falling edge clock transition as shown below.

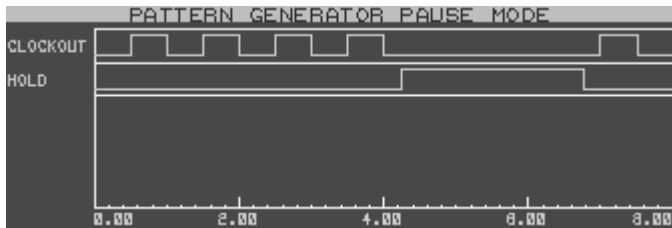


We are clocking internally at 1Hz. Note that the trigger takes place on the falling edge of the trigger pulse and that the pattern does not reset until the falling edge of the clock pulse subsequent to the trigger action.

External Hold

Holding the pattern in it's current state.

If you want to hold the pattern for a period of time you can do so by arranging for the *hold* pin to be high during the period in which you want to pause. Releasing the hold pin will restart the pattern synchronously if you are using an internal clock. That is, should the hold pin go high halfway through a clock cycle, then when you release the hold pin the next bit will be driven onto the output pins half a clock cycle later.



When the hold pin goes high the internal clock is paused. On release of the hold pin the clock resumes relative to the point in the clock cycle at which it was paused.

Additional Functionality

Loading and saving a pattern script.

Pattern Scripts can be loaded or saved by right clicking over the grid and selecting the relevant option from the popup menu. This is particularly useful if you want to use a particular pattern in several designs.

The pattern scripts are plain text and are simply a comma-separated list of bytes where each byte value represents a column on the grid. Any line beginning with a semi-colon is taken to be a comment line and is ignored by the parser. By default the byte format is hexadecimal though if you are creating your own script you can enter values in decimal, binary or hexadecimal.

Setting specific values for dials.

You can specify an exact value for both the bit and the trigger frequency by double clicking the mouse on the appropriate dials. This launches a floating edit box into which you can enter the value. By default the value entered is assumed to be a frequency but you can specify a value in seconds or fractions thereof by appending a suitable suffix to the value (sec, ms, etc.). Additionally, should you wish the trigger to be an exact multiple of the bit clock you can append the suffix 'bits' to the multiple desired (e.g. 5bits).

To confirm input press the *Enter* button or to cancel press the *Escape* key or click elsewhere on the Pattern Generator window.

These values can also be specified prior to simulation via the appropriate properties in the *Edit Component* dialogue form.

Setting specific values for the pattern grid.

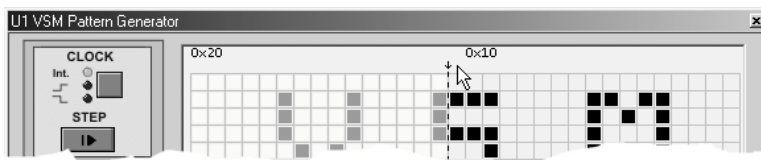
You can specify a specific value for any column on the grid by left clicking the mouse on the text displaying the current value for that column. A floating edit box is launched into which you can enter the desired value. You can specify the value in decimal (e.g. 135), hexadecimal (e.g. 0xA7) or binary (e.g. 0b10110101).

To confirm input press the *Enter* button or to cancel press the *Escape* key or click elsewhere on the Pattern Generator window.

For convenience you can set a column via the CTRL+1 keyboard shortcut and clear a column via the CTRL+SHIFT+1 keyboard shortcut with the mouse over the appropriate column.

Specifying the period length of the pattern manually.

You can specify a manual period by left clicking the mouse just above the grid itself at the column you want the pattern to finish on. To deselect the period right click the mouse once in the same area. This is shown below.



The mouse is pointing at the area where period selection can be specified. We can see that the period bar is indicating the pattern termination point and that everything to the left of the selection is faded out to indicate that the pattern will stop at the period bar.

Single stepping to advance the pattern.

The step button can be used to advance the simulation in time periods equal to that of the bitclock specified either internally or from an external clock. The simulation will run until the next clock cycle is completed and then suspend again.

Alternating the grid display mode.

The grid display can be switched between hexadecimal and decimal display modes. This can be done either by right clicking on the grid and selecting the desired option from the menu or alternatively by using the CTRL+X (Hexadecimal display) and CTRL+D (Decimal display) shortcut keys.

Specifying Output.

Edit the Schematic part for the Pattern Generator to produce the *Edit Component* dialogue form. The property at the bottom allows you to configure whether you want to output the pattern on both bus and pins, pins only or bus only.

Tooltip Display.

You can enable a tooltip which will follow the mouse indicating the current row and column information. This can be toggled on and off either via the context mouse on the right mouse button or through the shortcut key CTRL+Q. Note that tooltip mode is disabled during a Block set or clear.

Block Editing.

You can use the *Block Set* and *Block Clear* commands to help you quickly configure the grid to the desired pattern. These are accessible via the context menu on the right mouse button or via their shortcut keys (CTRL+S for Block set and CTRL+C for Block clear). Note that the Block Editing commands are disabled when in tooltip mode.

VSM USER INTERFACE ELEMENTS

Rotary Dials

The VSM Virtual Instruments use mouse operated rotary dials (knobs) for adjusting some parameters. The procedure for adjusting these is as follows:

To set a rotary dial:

1. Point somewhere inside the dial.
2. Press the left mouse button and hold it down.
3. Move the mouse pointer away from the dial and then around the centre of the dial, tracing out a circular arc to rotate the dial to the required setting.
4. The dial will track the angle subtended by the mouse pointer from its centre. The further away you move the mouse, the finer the degree of control that you will have.
5. Release the mouse button to fix the new position of the dial.

WORKING WITH MICROPROCESSORS

INTRODUCTION

The combination of mixed mode simulation and circuit animation becomes most potent when used in the context of micro-controller based systems. Many such systems involve a user interface, or at least complex sequences of external events that cannot easily be simulated in a non-interactive environment, and PROTEUS VSM was created primarily to address this problem. Consequently, a substantial chunk of its functionality centres around micro-processor development.

In particular, the editing and compiling of source code, is integrated into the design environment so that you can edit source code and observe the effects of your changes with maximum ease. A source file can be brought up for editing with two keystrokes and the simulation re-run from there with just two more.

SOURCE CODE CONTROL SYSTEM

Overview

The source code control system provides two major functions:

- To register source code files within ISIS so that they can be brought up for editing without manually switching to another application.
- To define the rules for compiling source code into object code. Once set up, these rules are carried out every time a simulation is performed so that the object code is kept up to date.

Note that it is not necessary to use the source code control system in order to simulate microprocessor based designs. Indeed, if your chosen assembler or compiler has its own IDE, you may prefer to work directly in that, and only switch to Proteus when you have produced an executable program. If you are planning to work this way, then see the following section *Using a 3rd Party IDE* on page 50.

Attaching Source Code to the Design

To add a source file to the design:

1. Select the *Add/Remove Source Files* command from the *Source* menu.
2. Select the *Code Generation Tool* for the source file. If you are planning to use a new assembler or compiler for the first time, you will need register it using the *Define Code Generation Tools* command.
3. Click the *New* button and select or enter a name for the source code file with the file selector. You can enter a filename textually if it does not yet exist.
4. Enter the flags required specifically for processing this source file in the *Flags* field. Flags required every time a particular tool is used can be entered when the tool is registered.
5. Click OK to add the source file to the design.

Don't forget to edit the micro-processor and assign the name of the object code file (usually a HEX file) to its **PROGRAM** property. ISIS can't do this automatically as you might have more than one processor on the diagram!

Working on your Source Code

To edit a source file:

1. Press ALT-S
1. Press the ordinal number of the source file on the source menu.

If you prefer to use a more advanced text editor, see page 50.

To switch back to ISIS, build the object code and run the simulation:

1. From the text editor, press ALT-TAB to switch back to ISIS.
2. Press F12 to execute, or CTRL-12 to start debugging.

Either way, ISIS will instruct the text-editor to save its files, examine the time-stamps of the source code and object code files, and invoking the appropriate code generation tools to build the object code.

To rebuild all the object code:

1. Select the *Build All* command from the source menu.

ISIS will execute all the code generation tools required to build the object code, irrespective of the time/date stamps of the object code files. The command line output from the tools will be

displayed in a window. This provides an excellent way to check that everything has built without errors or warnings.

Installing 3rd Party Code Generation Tools

A number of shareware assemblers and compilers can be installed into the TOOLS directory from the system CD, and these will be set up automatically as code generation tools by the PROTEUS setup program. However, if you want to use other tools, you will need to use the *Define Code Generation Tools* command on the *Source* menu.

To register a new code generation tool:

1. Select the *Define Code Generation Tools* command from the *Source* menu.
2. Click *New* and use the file selector to locate the executable file for the tool. You can also register batch files as code generation tools.
3. Enter extensions for the source and object code files. These determine the file types that ISIS will look for when deciding whether to execute the tool for a particular source file, or not. If you check *Always Build* then the tool is always invoked, and an object code extension is not required.
4. Specify a proforma command line for the tool. Use %1 to represent the source code file and %2 for the object code file. You can also use %\$ for the path to the PROTEUS directory and %~ for the directory in which the DSN file is located.

This is a good place to put command line flags that are needed to make tools run quietly (i.e. without pausing for user input), and to specify paths to include directories of processor specific header files etc.

If you want to use the source level debugging features of PROTEUS VSM, you will need a *Debug Data Extractor* for your assembler or compiler. This is a small command line program, which extracts object code/source line cross reference information from the list file produced by the assembler or compiler. We hope to support all popular assemblers and compilers, so check our website for the latest info.

If you have a DDX program, enter the extension for the list file or symbolic debug file that is produced by the code generation tool, and click *Browse* to select the path and filename of the DDX program.

Using a MAKE Program

In some cases, the simple build rules implemented by ISIS may be insufficient to cater for your application – especially if you project involves multiple source and object code files, and a

linking phase. In such cases, you will need to use an external 'MAKE' program, and you can proceed as follows:

To set up a project with an external make program:

1. Install your make program (usually supplied with the assembler/compiler environment) as a code generation tool. Set the source extension to MAK and check *Always Build*. For a typical make program, set the command line proforma as:

-f %1
2. Use the *Add/Remove Source Files* command to add the makefile (e.g. MYPROJECT.MAK) to the design.
3. Also add the source code files, but select the *Code Generation Tool* as <NONE>

Each time the project is built, ISIS will run the external MAKE program, with the project's makefile as a parameter. It is then up to the make program to decide what code generation tools are run. A good make program will provide enormous flexibility.

Using a 3rd Party Source Code Editor

PROTEUS VSM is supplied with a simple text editor – SRCEDIT which you can use to edit source code files. SRCEDIT is essentially a modified version of NOTEPAD, which can open multiple source files, and can respond to a DDE request to save any modified buffers.

If you have a more advanced text editor, such as *UltraEdit*, you can instruct ISIS to use this instead. Note that IDE type environments probably won't respond to DDE commands and so are unsuitable for integration in this way. However, you could always ask the vendor to add this support – it's not hard.

To set up an alternative source code editor:

1. Select the *Setup External Text Editor* command from the *Source* menu.
2. Click the *Browse* button and use the file selector to locate the executable file of your text editor.
3. ISIS instructs the text editor to open and save files using a DDE protocol. Refer to the text editor's documentation or supplier for details of the command syntax. If unsure of the service name, try the name of the product – e.g. ULTRAEDIT.

USING A 3RD PARTY IDE

Most professional compilers and assemblers come with their own integrated development environment or IDE. Examples include IAR's Embedded Workbench, Keil's uVision 2, Microchip's MP-LAB and Atmel's AVR studio. If you are developing your code with one of

these tools, you may find it easier to carry out the editing and compilation steps within the IDE and then switch to Proteus VSM only when you have produced an executable image (e.g a HEX or COD file) and are ready to simulate it.

Proteus VSM supports two ways of working with an external IDE

- Using Proteus as an external debugger - control of the debugging session is carried out from within ISIS, much as you would for a normal CAD simulation.
- Using Proteus as a plug-in simulator. - control of the debugging session is carried out from within the IDE's debugger. Proteus acts as a kind of virtual In Circuit Emulator, communicating with the IDE over TCP/IP. In this mode, you can run your IDE's debugger on one PC and the Proteus simulation on the other.

Using Proteus VSM as an External Debugger

To Proteus as an external debugger requires that the symbolic debug format produced by your compiler is supported by one of the *loaders* available in Proteus. A loader extracts both the addresses of each source line in the high level language program, and - where possible - the locations of the program variables.

Common debug formats are COD (used in the PIC world), UBROF - used by all IAR's compilers, and OMF (used in 8051 circles). We also provide loaders for other proprietary formats such as the list files produced Crownhill PICBasic. See our the support area of our website under "*3rd Party Compilers*" for up to date information.

Assuming that there is a loader for your chosen compiler, the procedure for loading the program into the simulated micro-processor is extremely simple:

To load a program produced in an external IDE

1. Ensure that the program is compiled and linked with no errors.
2. Edit the **PROGRAM** property of the CPU model to be the name of the executable image file produced by the compiler or linker, e.g. MYPROG.COD.

Do not enter the name of the source file - Proteus VSM does not simulate 'C' or 'ASM' files; the CPU models load and execute binary machine code.

3. Press the PLAY button on the animation control panel to begin real-time simulation or press the STEP button to run up to the first source level instruction. In the latter case, the *Source Window* will appear and you can commence stepwise debugging of your code.

Using Proteus VSM as a Virtual In-Circuit Emulator (ICE)

At the time of writing, only Keil uVision 2 for the 8051 supports the virtual ICE approach although we are working with IAR, Microchip and Atmel amongst others to integrate Proteus VSM with their tools. It is a fast moving area and we would recommend you check our the support area of our website under "*3rd Party Compilers*" for the latest information and documentation. Instructions for using setting up *uVision2* to work with Proteus will also be found there.

POPUP WINDOWS

Most micro-processor models written for PROTEUS VSM will create a number of popup windows which can be displayed and hidden from the *Debug* menu. These windows come in three major types:

- Status Windows – a processor model will generally use one of these to display its register values.
- Memory Windows – typically there will be one of these for each memory space in the processor architecture. Memory devices (RAMs and ROMS) also create these windows.
- Source Code Windows – one of these will be created for each processor on the schematic.

To display a popup window:

1. Start debugging mode by pressing CTRL-F12, or if it is already running, click the *Pause* button on the *Animation Control Panel*.
2. Press ALT-D and then the ordinal number of the required window on the *Debug* menu.

These window types can only be displayed when the simulation is paused and are hidden automatically when it is running to give you better access to the active components on the schematic itself. When the simulation is paused (either manually, or due to a breakpoint) the windows that were showing will re-appear.

The debug windows all have right button context menus – if you point at the window and click the right mouse button, a menu will appear from which you can control the appearance and formatting of the data within the window.

The positions and visibility of the debug windows are saved automatically to a PWI file with the same filename as the current design. The PWI file also contains the locations of any breakpoints that have been set, and the contents of the watch window.

SOURCE LEVEL DEBUGGING WITHIN PROTEUS VSM

Overview

PROTEUS VSM supports source level debugging through the use of *debug loaders for supported* assemblers and compilers. The current set of debug loaders are contained within the system file LOADERS.DLL, and the number tools that are supported is increasing quite rapidly. The latest information is available from the support area of our website under "*3rd Party Compilers*".

Assuming that you have used a supported assembler or compiler, PROTEUS VSM will create a source code window for each source file used in the project, and these windows will appear on the *Debug* menu.

The Source Code Window

The source code window has a number of features:

- A blue bar represents the current line number, at which a breakpoint will be set if you press F9 (*Toggle Breakpoint*), and to which the program will execute if you press CTRL-F10 (*Step To*).
- A red chevron indicates the current location of the processor's program counter.
- A red circuit marks a line on which a breakpoint has been set.

The right hand context menu provides a number of further options including: *Goto Line*, *Goto Address*, *Find Text* and toggles for displaying line numbers, addresses and object code bytes.

Single Stepping

A number of options for single stepping are provided, all available from the toolbar on the source window itself or from the *Debug* menu.

- Step Over – advances by one line, unless the instruction is a sub-routine call, in which case the entire subroutine is executed.
- Step Into – executes one source code instruction. If no source window is active, it executes one machine code instruction. These are usually the same thing anyway unless you are debugging in a high level language.
- Step Out – executes until the current sub-routine returns.
- Step To – executes until the program arrives at the current line. This option is only available when a source code window is active.

Note that apart from *Step To*, the single stepping commands will work without a source code window. It is possible – although not so easy – to debug code generated by a tool for which there is no loader support.

Using Breakpoints

Breakpoints provide a very powerful way to investigate problems in the software or software/hardware interaction in a design. Typically, you will set a breakpoint at the start of a subroutine that is causing trouble, start the simulation running, and then interact with the design such that the program flow arrives at the breakpoint. At this point, the simulation will be suspended. Thereafter you can single step the program code, observing register values, memory locations and other conditions in the circuit as you go. Turning on the *Show Logic State of Pins* effect can also be very instructive.

When a source code window is active, breakpoints can be set or cleared on the current line by pressing F9. You can only set a breakpoint on a line which has object code.

If the source code is changed, PROTEUS VSM will endeavour to re-located the breakpoints based on sub-routine addresses in the file, and by pattern matching the object code bytes. Obviously, if you change the code radically this can go awry but generally it works very well and you shouldn't need to think about it.

Variables Window

Most of the loaders provided with Proteus VSM are able to extract the location of the program variables as well as the source line number addresses. Whenever this is possible, Proteus displays a variables window as well as a source window.

There are a number of points to note about the variables window:

- When single stepping, any variables that change value are highlighted.
- The format in which each variable is displayed can be adjusted by clicking right on it and choosing an alternative format from the context menu.
- Although the variables window is hidden whilst the program is running, you can drag & drop variables to the watch window where they will remain visible.
- Depending on how the compiler re-uses memory, local variables which out of scope variables may display invalid values.

THE WATCH WINDOW

Whereas the memory and register windows belonging to processor models are only displayed when the simulation is paused, the *Watch Window* provides a means to display values that are updated in real time. It also provides a means to assign names to individual memory locations which can be more manageable than trying to find them in a memory window.

To add an item to the Watch Window:

1. Press CTRL-F12 to start debugging, or pause the simulation if it is already running.
2. Display the memory window containing the item to be watched, and the watch window itself using the numbered options on the *Debug* menu.
3. Mark a memory location or range of memory locations using the left mouse button. The selected range should appear in inverted colours.
4. Drag the selected item(s) from the memory window to the watch window.

You can also add items to the *Watch Window* by using the *Add Item* command on its right mouse button context menu.

Modifying Items in the Watch Window

Having got an item or items in into the window, you can now select an item with the left mouse button and then:

- Rename it by pressing CTRL-R or F2.
- Change the data size to any of the options on the right mouse button context menu. For data sizes that comprise several bytes (e.g. 16 or 32 bit words, or strings) the second and subsequent bytes are assumed to follow on from the item address. Consequently, to display a multi-byte word or string you only need to drag in the first byte from the memory window.
- Change the number format to binary, octal, decimal or hex.

BREAKPOINT TRIGGER OBJECTS

Overview

A number of component objects are provided which trigger a suspension of the simulation when a particular circuit condition arises. These are especially useful when combined with the single stepping facilities, since the circuit can be simulated normally until a particular condition arises, and then single stepped in order to see exactly what happens next.

The breakpoint trigger devices may be found in the REALTIME device library.

Voltage Breakpoint Trigger – RTVBREAK

This device is available in 1 and 2 pinned forms, and triggers a breakpoint when the voltage at its single pin, or the voltage between its two pins is greater than the specified value. You can couple this device to an arbitrary controlled voltage source (AVCS) to trigger a breakpoint on complex, formula based conditions involving multiple voltages, currents etc.

Once the trigger voltage has been exceeded, the device does not re-trigger until the voltage has dropped below the trigger threshold and risen again.

Current Breakpoint Trigger – RTIBREAK

This device has two pins and triggers a breakpoint when the current flowing through it is greater than the specified value.

Once the trigger current has been exceeded, the device does not re-trigger until the current has dropped below the trigger threshold and risen again.

Digital Breakpoint Trigger – RTDBREAK

This device is available with various numbers of pins, and you can make other sizes yourself if you need to. It triggers a breakpoint when the binary value at its inputs equals the value assigned to the component. For example, specifying the value of an RTDBREAK_8

0x80

will cause a it to trigger when D7 is high and D0-D6 are low.

Once the trigger condition has arisen, the device does not re-trigger until the voltage a different input value has been seen.

GRAPH BASED SIMULATION

INTRODUCTION

Although interactive simulation has many advantages, there are still some situations in which it is advantageous to capture the entire simulation run to a graph and study the results at your leisure. In particular, it is possible to zoom in on particular events within the simulation run and take detailed measurements. Graph based simulation is also the only way to perform analyses which do not take place in real time at all, such as small-signal AC analysis, Noise Analysis and swept parameter measurements.

Graph based simulation is not available in PROTEUS Lite.

SETTING UP A GRAPH BASED SIMULATION

Overview

Graph based simulation involves five main stages. These are summarized below, with detailed explanations of each stage being given in the subsequent sections:

- Entering the circuit to be simulated.
- Placing signal generators at points that require stimulus and test probes at points that you wish to inspect.
- Placing a graph corresponding with the type of the analysis you wish to perform - e.g. a frequency graph to display a frequency analysis.
- Adding the signal generators and test probes to the graph in order to display the data they generate/record.
- Setting up the simulation parameters (such as the run time) and executing the simulation.

Entering The Circuit

Entering the circuit you want to simulate is exactly the same as entering any other design in to ISIS; the techniques for this are covered in detail in the ISIS manual itself.

Placing Probes and Generators

The second stage of the simulation process is to set up signal generators at points that require stimuli and probes at points of interest. Setting up signal generators and probes is extremely easy as they are treated like other ISIS objects such as components, terminals, or

junction dots. All that is required is to select the appropriate icon, pick the type of generator or probe object from the selector, and place it on the schematic where you want it. This can be either by placing it directly on to an existing wire, or by placing it like a component and then wiring to it later. Defining the signal produced by a generator is then just a matter of editing the object and entering the required settings on its dialogue form.

At this stage you can also isolate a section of the design so that only certain components are involved in the simulation. This is achieved by checking the *Isolate Before* and *Isolate After* options in the probes and generators. The use of probes and generators in this way provides not only for faster simulations, but also means that errors do not creep in from removed wires being forgotten and never replaced.

Placing Graphs

The third stage of the simulation process is to define what analysis type or types you want performed. Analysis types include analogue and digital transient analyses, frequency analyses, sweep analyses, etc. Within ISIS, defining an analysis type is synonymous with placing a graph object of the required analysis type. Again, as graphs are just like most other objects within ISIS, placing one is simply a case of selecting the correct icon, selecting the required graph type, and placing the graph on the design, alongside the circuit. Not only does this allow you to view several types of analysis simultaneously, it fits very well with the 'drag-and-drop' methodology adopted throughout ISIS and has the added benefit that you can view (and generate hard-copy output with) the graphs alongside the circuit that generated them.

Adding Traces To Graphs

Having placed one or more graph objects, you must now specify which probe/generator data you want to see on which graphs. Each graph displays a number of *traces*. The data for a trace is generally derived from a single probe or generator. However, ISIS provides for a trace to display the data from up to four separate probes/generators combined by a mathematical *Trace Expression*. For example, a trace might be set up to display the product of the data from a voltage probe and current probe (both monitoring the same point) so effectively displaying the power at the monitored point.

Specifying the traces to be displayed on a graph can be done in a number of ways: you can tag-and-drag a probe onto a graph or you can tag several probes/generators and add them all to a graph in a single operation, or for traces requiring trace expressions, you can must use a dialogue form to select the probes and specify the expression.

The Simulation Process

Graph based simulations is *Demand Driven*. This means that the emphasis is on setting up generators, probes and graphs in order to determine what you want to measure, rather than on setting up the simulator and then running some kind of post processor in order to examine the results. Any parameters that are specific to a given simulation run are specified by either editing the explicit properties of the graph itself (e.g. the start and stop times for the simulation run) or by assigning additional properties to the graph (e.g. for a digital simulation, you can pass a 'randomise time delays' property to the simulator).

So what happens once you have initiated a simulation? In brief, the action proceeds through the following steps:

- Netlist generation - this is the usual process of tracing wires from pin to pin, and producing a list of components and a list of pin to pin connection groups, or *nets*. In addition, any components in the design that are to be simulated by *model files* are replaced by the components contained in these files.
- Partitioning - ISIS then looks at the points you where you have placed probes and traces back from these to where you have injected signals. This analysis results in the establishment of one or more partitions that may need to be simulated and the order in which they must be simulated. As each partition is simulated, the results are stored in a new *partition file*.
- Results Processing - Finally, ISIS accesses the partition files to build up the various traces on the graph. The graph is then re-displayed and can be maximized for measurement taking and so forth.

If errors occur during any of these stages, then the details are written to the simulation log file. Some errors are fatal, and others are just warnings. In the case of fatal errors, the log file is displayed for immediate viewing. If only warnings have occurred then the graph is re-displayed and you are offered the choice of viewing the log if you want to. Most errors relate either to badly drawn circuits (which cannot, for one reason or another, be mathematically solved), or to the omission or incorrect linking of model files.

GRAPH OBJECTS

Overview

A *graph* is an object that can be placed on to the design. The purpose of a graph is to control a particular simulation and display the results of that simulation. The type of analysis type performed by the simulation is determined by the type of the graph placed. The part of the design that is simulated and the data that is displayed on the graph is determined by the probe and generator objects that have been added to the graph.

The Current Graph

All graph-specific commands are on the *Graph* menu. The lower portion of this menu also contains a list of all the graphs in the design, with the *current* graph being marked by a small marker at the left of its name. The *current* graph is the last graph that was simulated or acted upon by a command.

Any command from the *Graph* menu can be invoked for a specific graph by pointing at the graph with the mouse and using the command's keyboard short-cut (shown on the menu to the right of the command). If the mouse pointer is not pointing at a graph, or if you select the command directly from the menu, the command is invoked for the *current* graph.

Placing Graphs

To place a graph:

1. Obtain a list of graph types by selecting the the *Graph* icon. The list of graph types is displayed in the selector on the right hand side of the display.
2. Select the type of graph you wish to place from the *Object Selector*.
3. Place the mouse in the *Editing Window* at the point you wish the top left corner of the graph to appear. Press down the left mouse button and drag the out a rectangle for the size of the graph you wish to place, then release the mouse button.

Editing Graphs

All graphs can be moved, resized or edited to change their properties using the standard ISIS editing techniques.

The properties of a graph are changed via its *Edit...* dialogue form invoked as for any object in ISIS by either first tagging it and then clicking the left mouse button on it, or by pointing at it with mouse and pressing CTRL+'E' on the keyboard.

Adding Traces To A Graph

Each graph displays one or more *traces*. Each trace normally displays the data associated with a single generator or probe. However, for analogue and mixed graph types it is possible to set up a single trace to display the data from between one and four probes combined by a mathematical formula which we call a *trace expression*.

Each trace has a label that is displayed alongside the y-axis to which the trace is assigned. Some graph types have only one y-axis and there is no option to assign the trace to a particular y-axis. By default, the name of a new trace is the same as the name of the probe (or the first probe in the trace's expression) - this can be changed by editing the trace.

Traces can be defined in one of three ways:

- Dragging and dropping an individual generator or probe on to the relevant graph.
- Tagging a selection of probes and using the *Quick Add* feature of the *Add Trace* command.
- Using the *Add Trace* command's dialogue form.

The first two methods are quick to use but limit you to adding a new trace for each generator or probe assigned to the graph. The third method is somewhat more involved but gives greater control over the type of trace added to the graph. In particular, traces that require *trace expressions* must be added to the graph via the *Add Trace* command's dialogue form.

To drag-and-drop a probe or generator on to a graph:

1. Tag the generator or probe you wish to add to a graph using the right mouse button.
2. Click and hold down the left mouse button on the generator or probe and drag the probe over to the graph and release the mouse button.

A new trace is created and added to the graph; the new trace displays the data associated with the individual probe/generator. Note that any existing traces may be shrunk in order to accommodate the new trace.

For graphs with two y-axes, releasing the probe or generator on left or right half of the graph assigns the new trace to the respective axis. Furthermore, for the mixed analogue and digital transient analysis graph type, releasing the generator or probe over existing digital or analogue traces creates a new trace of the respective type (the first probe or generator released on a mixed graph type is always creates an analogue trace).

To quick-add several generators or probes to a graph:

1. Ensure the graph you wish to add the generators or probes is the *current* graph. The current graph is the graph whose title is shown selected on the *Graph* menu.
2. Tag each generator or probe you want to add to the graph.
3. Select the *Add Trace* command on the *Graph* menu. As a result of the tagged probes and generators, the command first displays a *Quick add tagged probes?* prompt.
4. Select the *Yes* button to add the tagged generators and probes to the current graph.

A new trace is created for each tagged generator or probe and is added to the graph in alphabetical order; each new trace displays the data associated with its associated individual generator or probe. Traces are always assigned to the left y-axis for graph types that support two axes.

The Add Trace Command Dialogue Form

Assuming you are not performing *Quick Add*, the *Add Trace* command will display an *Add Trace* dialogue form (each graph type has slight variations). This form allows you to select the name of the new trace, its type (analogue, digital, etc.), the y-axis (left or right) to which it is to be added, up to four generator or probes whose data it is to use, and an expression that combines the data of the selected generators and probes.

To add a trace to a graph using the *Add Trace* command

1. Invoke the *Add Trace* command on the *Graph* menu against the graph to which you want to a new trace.

If there are any tagged probes or generators, a *Quick add tagged probes?* prompt will be presented. Respond by selecting the *No* button.

The *Add Transient Trace* dialogue form is displayed.

2. Select the type of trace you wish to add to the graph by selecting the appropriate *Trace Type* button. Only those trace types applicable to the graph are enabled.
3. Select the y-axis to which the new trace is to be added. Only those axes applicable to the graph and trace type selected are enabled.
4. Select one of the *Selected Probes* buttons *P1* through to *P4*, and then select a probe or generator from the *Probes* list box to be assigned to the selected trace probe. The name of the selected generator or probe will appear alongside the *Selected Probes* button and the *Selected Probe* name (*P1* through *P4*) will appear in the *Expression* field if it is not already present.

If a trace expression is not allowed, only the *P1* trace will be enabled - and the new trace will display the data associated with the *P1* probe selected.

5. Repeat steps [3] & [4] until you have selected all the generators or probes you require for the trace.
6. Enter the *trace expression* in the *Expression* field. Within the expression, the selected probes should be represented by the names *P1*, *P2*, *P3* and *P4* which correspond to the probes selected alongside the **P1**, **P2**, **P3** and **P4** buttons respectively.

If the trace type selected does not support a *trace expression* the contents of the *Expression* field will be ignored.

7. Select the OK button to add the new trace to the graph.

Editing Graph Traces

An individual trace on a graph can be edited to change its name or expression.

To tag, edit and untag a graph trace:

1. Ensure the graph displaying the trace to be edited is *not* tagged.
2. Tag the *trace* by clicking the right mouse button on the trace's name. A tagged trace displays its name highlighted.
3. Click the left mouse button on the trace's name. The trace displays its *Edit Graph Trace* dialogue form.
4. Edit the trace's name or expression as required. For a trace type that does not support trace expressions any changes to the *Expression* field will be ignored.
5. Select the OK button to accept the changes.
6. Untag the trace by clicking the right mouse button over the graph, but not over any trace names.

Changing the Order and/or Colour of Graph Traces

The order of the traces on a graph can be adjusted by dragging the labels with the left mouse button. The trace can be tagged or otherwise, as you wish.

- For digital traces the purpose of moving them is simply that of obtaining a particular stacking sequence on the graph.
- For an analogue graph, you can both drag the traces from the left to right axis and back. You can also change the colour assignments, which are made on the basis of the stack order, by shuffling the vertical stacking order.

The actual colours assigned to the sequence positions can be re-configured by maximising a graph (any one will do) and then using the *Set Graph Colours* command on the *Template* menu.

THE SIMULATION PROCESS

Demand Driven Simulation

When we designed Proteus, one of our major goals was to make the use of simulation software much more intuitive than it has previously been. Much of the problem with previous simulation packages stems from the fact that number crunching parts were written first and other aspects such as displaying the results graphically were added on later, almost as an afterthought. This tends to result in a fragmented way of working in which you run one

program to draw the schematic, another to actually simulate the circuit, and yet another to display the results.

Proteus is very different in that, having drawn the circuit, you start by answering the question of what you want to see by means of placing and setting up a graph. This graph then persists until you delete it, and each time you want to see the effect of a change to the circuit, you just update the graph by pointing at it and pressing the space-bar. We call this *Demand Driven Simulation* since ISIS must work out from the graph what actually requires simulating, rather than you having to do it textually. Furthermore, you can place several graphs which perform different experiments with each one 'remembering' a different setup for the simulators.

The most important benefit of all comes from the fact that a given graph defines a set of points of interest in the design by virtue of the probes that have been assigned to its traces. ISIS is then able to use this information to figure out which parts of the design actually need to be simulated, rather than you having to calve the design up manually. It follows that a *complete* design, ready for PCB layout, can be simulated without massive editing, even if you only want to simulate particular parts of it.

A further benefit arises from the fact that there is no risk of forgetting to undo modifications made for simulation purposes because there are none to undo; the probe and generator gadgets that you do place are ignored when generating netlists for PCB layout.

Executing Simulations

Once a graph has been placed with probes and generators assigned to it, you can initiate a simulation by pointing at the graph and pressing the space-bar. ISIS then determines which parts of the design need to be simulated in order to update the graph, runs the simulations, and displays the new data.

Throughout the simulation process, a *simulation log* is maintained. In general, the log contains little of interest. However, if simulation errors occur, or you have requested the simulation netlist to be logged (see Editing Graphs on page 60 on editing graphs for how to do this) or the analysis type results in data that is not amenable to graphical display (e.g. an analogue Noise analysis) then you *will* need to view the log at the end of the simulation run.

To update a graph and view the simulation log:

1. Either ensure the graph you want to update is the *current* graph and then select the *Simulate* command from the *Graph* menu, or, place the mouse pointer over the graph you wish to update and press the space bar.
2. At the end of the simulation, if errors have occurred, you will be prompted to *Load partial results?* If you reply *YES* simulation data is loaded up to the time of the error and displayed on the graph. If you reply *NO* the simulation log is displayed in a pop-up text viewer window.

If no errors occurred, or if you selected *YES* above, you can still view the simulation log by either selecting the *View Log* command from the *Graph* menu or by using its keyboard short-cut, CTRL+V.

What Happens When You Press the Space-Bar

When you update a graph, either using the *Simulate* command on the *Graph* menu or its space-bar keyboard short-cut - a great deal of analysis and processing occurs over and above the actual analogue or digital analyses themselves. We now outline in brief various steps in this process:

- Netlist compilation - this is the usual process of tracing wires from pin to pin, and producing a list of components and a list of pin to pin connection groups, or *nets*. The resulting netlist is held in memory at this stage.
- Netlist linking - some components are modelled using sub-netlists or *model files* which are generally kept in the directory specified by the *Module Path* field of the *Set Paths* command's dialogue form. Many models are supplied with Proteus and you can, of course, create your own as well. You can think of a model as being a child sheet of a hierarchical design in which the parent object is the component to be modelled.

Each time a component modelled in this way is encountered, the original part is removed from the netlist and replaced by the contents of the model file, with the inputs and outputs of the model connected in where the original component's pins were.

At the end of this process, every component left in the netlist should have a **PRIMITIVE** property, which means that it can be directly simulated by PROSPICE.

- Partitioning - ISIS then looks at the points you where you have placed probes and traces back from these to where you have injected signals. A signal is deemed to be injected by any of: a power rail, a generator with its *Isolate Before* flag set or tape output. This analysis results in the establishment of one or more partitions that may need to be simulated.

Further analysis then works out the *order* in which they must be simulated. For example, if partition A has an output that connects to an input of partition B then clearly A must be simulated first. If it turns out that B has an output driving A as well, then all is lost - it is up to you to use tape objects in such a way that cyclic dependencies do not occur.

Yet more analysis then establishes whether there are any existing results in the directory specified by the *Results Path* field (as set by the *Set Paths* command on the *System* menu) that relate to identical simulation runs for any of the current set of partitions. If there are, and if the partition concerned is not driven by a partition which is itself being re-simulated, then ISIS does not bother re-simulating the partition, but uses the existing

results instead. It follows that if you are working on the back end of a design, there is no need to keep re-simulating the front end.

You will appreciate that this part of the system is extremely clever!

- Simulator Invocation - ISIS now invokes PROSPICE on each partition in turn to carry out the actual simulation.

Each simulator invocation results in a new partition file, and progress information is also added to the simulation log which is maintained throughout the entire simulation process.

- Results Processing - Finally, ISIS accesses the partition files to build up the various traces on the graph. The graph is then re-displayed and can be maximised for measurement taking and so forth.

If errors occur during any of these stages then the details are written to the simulation log. Some errors are fatal, and others are just warnings. In the case of fatal errors, the simulation log is displayed for immediate viewing. If only warnings have occurred then the graph is re-displayed and you are offered the choice of viewing the simulation log if you want to. Most errors relate either to badly drawn circuits (which cannot, for one reason or another, be mathematically solved), or to the omission or incorrect linking of model files.

ANALYSIS TYPES

INTRODUCTION

There are thirteen types of graph available; each one displays the results of a different type of circuit analysis supported by PROSPICE:

Analogue	<p>Plots voltages, and/or currents against time, much like an oscilloscope. Additionally, expressions involving several probed values can be plotted - for example, a current can be multiplied by a voltage to give a plot of instantaneous power.</p> <p>This mode of analysis is often referred to as <i>Transient Analysis</i>.</p>
Digital	<p>Plots logic levels against time, much like a logic analyser. Traces can represent single data bits or the binary value of a bus.</p> <p>Digital graphs are computed using <i>Event Driven Simulation</i>.</p>
Mixed Mode	<p>Combines both analogue and digital signals on the same graph.</p>
Frequency	<p>Plots small signal voltage or current gains against frequency. This is also known as a Bode plot, and both magnitude and phase can be displayed. In addition, with the use of <i>Trace Expressions</i> you can produce input and output impedance plots.</p> <p>Note that plotted values are gains referenced to a specified input generator.</p>
DC Sweep	<p>Steady state operating point voltages or currents against a sweep variable. As with <i>Analogue</i> analysis, expressions combining several probed values can also be plotted.</p>
AC Sweep	<p>Creates a family of frequency plots with one response for each value of the sweep variable.</p>
Transfer	<p>Plots characteristic curves or curve families by sweeping the value of one or two input generators and plotting steady state voltages and currents. The value of the first generator variable is plotted on the x-axis; a separate curve is produced for each value of the second variable.</p>
Noise	<p>Input or Output Noise voltages against frequency. Also produces a listing of individual noise contributions.</p>

Distortion	Plots the level of 2 nd and 3 rd harmonic distortion against frequency. Can also be used to plot intermodulation distortion.
Fourier	Shows the harmonic content of a transient analysis. This is much like connecting a spectrum analyser in place of an oscilloscope.
Audio	Performs a transient analysis and then plays the result through your sound card. Can also generate Windows WAV files from circuit output.
Interactive	Performs an interactive simulation, and displays the results on a graph. This analysis type lets you combine the advantages of interactive and graph based simulation.
Conformance	Performs a digital simulation and then compares the results against a set of results stored from a previous run. This is especially relevant in creating software test suites for micro-controller based applications.

In addition, all types of analysis start by computing the operating point - i.e. the initial values of all node voltages, branch currents and state variables as at time 0. Information regarding the operating point is available from within ISIS via a point and shoot interface.

ANALOGUE TRANSIENT ANALYSIS

Overview

This graph type represents what you might expect to see on an oscilloscope. The x-axis shows the advance of time, whilst the y-axis displays voltage or current. We often refer to this type of analysis as *Transient Analysis* because it takes place in the time domain.

Transient analysis is perhaps the most commonly used form of analysis. Everything that you would measure with an oscilloscope on a real prototype can be measured on an analogue graph. It can be used to check that the circuit operates in the expected manner, to take quick measurements of gain, to assess visually the way in which a signal is distorted, to measure the current flowing from the supply or through individual components, and so on.

Method of computation

Technically speaking, this type of simulation can be referred to as *Non-Linear Nodal Analysis*, and is the form of computation used by all SPICE simulators. Considering a single point in time, every component in the circuit is represented as a combination of current sources and/or resistors. This arrangement can then be described as a set simultaneous equations using Ohms law and Kirchoff's laws, and the equations solved by Gaussian

Elimination. Each time the equations are solved, the values of the current sources and resistors are adjusted by laws built into the component models and the process is repeated until a stable set of values results.

For a simulation involving the advance of time there are two separate stages. The first task is to compute the *operating point* of the circuit - that is the voltages around the circuit at the very start of the simulation. This is then followed by considering the effect of advancing time on the circuit, and re-calculating the voltages at every step (iterating to convergence as described above). The size of each time increment is crucial to the stability of the calculations, and so PROSPICE will adjust it automatically within user defined limits. Circuits that are changing quickly, such as high switching speed line drivers, need smaller time steps and so require more effort to simulate than circuits that change more smoothly. The use of time step control and iterative solutions all adds up to a great deal of calculations, which can make transient analysis seem comparatively slow.

Since the algorithm involves iteration, there is always a possibility that the solution will not converge. Most commonly this happens at the initial time-point, meaning that the simulator cannot establish a stable or unique set of values for the operating point. Occasionally, however, it can occur further into a simulation where the circuit behaviour at a particular time becomes highly unpredictable. PROSPICE implements a number of techniques to help avoid such problems, but it is not impossible to defeat. This said, being based on Berkeley SPICE3F5 it is as good as you are going to get.

Using Analogue Graphs

Analogue graphs may have either just the left y-axis, or both left and right y-axes. Probes can be placed on a particular axis in two ways:


- By tagging and dragging the probe to the appropriate side of the graph.
- By using the *Add Transient Trace* dialogue form.



See Adding Traces To A Graph on page 60 for more information about adding traces to graphs.

It is often convenient to use the left and right y-axes to separate traces with different units. For example, if both voltage and current probes are displayed, then the left side can be used for voltage and the right side for current. It is possible to place a digital probe on an *Analogue* graph, but a *Mixed* graph is usually more appropriate.


To perform a transient analysis of an analogue circuit:

1. Add generators as necessary around the circuit, to drive the circuit inputs.
-  See Placing Generators on page 94 for information on placing generators, and GENERATORS AND PROBES on page 93 for a discussion of the different types of analogue generators.
2. Place probes around the circuit at points of interest. These may be at points within the circuit, as well as obvious outputs.
3. Place an *Analogue Graph*.
4. Add the probes (and generators, if desired) to the graph.
5. Edit the graph (point at it and press CTRL+'E', and set up the simulation stop time required, as well as labelling the graph and setting control properties if required.
6. Start the simulation by either selecting the *Simulate* command on the *Graph* menu or pressing the space bar.

Defining Analogue Trace Expressions

You will normally want to add voltage and current probes to your graphs, to see the operation of a circuit. However, in an analogue transient analysis it is possible to create traces that are mathematical *expressions* based on probe values. For example, suppose a circuit output has both a voltage and current probe placed on it. By multiplying these values together, the output power at this point will be plotted.

To plot the output power:

1. Add a current probe and a voltage probe to the circuit output.
 2. Invoke the *Add Transient Trace* dialogue form (press CTRL+'A') for the graph.
 3. Assign the voltage probe to **P1**.
 4. Assign the current probe to **P2**.
 5. Alter the expression to read **P1*P2**.
 6. Click on the **OK** button.
 7. Press the space bar to invoke the simulation.
-  See The Add Trace Command Dialogue Form on page 62 for a detailed description of the *Add Transient Trace* dialogue form.

DIGITAL TRANSIENT ANALYSIS

Overview

Digital graphs display what you would normally expect to see on a logic analyser. The x-axis shows the advance of time whilst y axis shows a vertical stack of signals. These can be either single data bits, or a representation of the binary values carried by a bus.

Method of Computation

Digital transient analysis is computed using a technique known as *Event Driven Simulation*. This is different from analogue transient analysis in that processing only occurs when some element of the circuit changes state. In addition, only discrete logic levels are considered and this enables component functionality to be represented at a far higher level. For example, we can think of a counter in terms of a register value that increments by one each time it is clocked, rather than in terms of several hundred transistors. These make event driven simulation several orders of magnitude faster than analogue simulation of the same circuit..

The Boot Pass

The purpose of the boot pass is to define the initial states of all nets in the circuit, prior to the simulation proper. The boot pass is carried out as follows:

- All input pins connected to the VCC and/or VDD nets are deemed to be high.
- All input pins connected to the GND and/or VSS nets are deemed to be low.
- All input pins connected to a net to which a generator is connected are deemed to be at the same state as the **INIT** property of the generator.
- All remaining pins are deemed to be initially floating.
- All models are requested (in no set order) to evaluate their inputs and set their output pins accordingly. As each output pin is set, the state of the net to which the pin connects is re-evaluated.
- As nets change state, models connected to them are asked to re-evaluate their outputs. This process continues until a steady state is found.

The Event Processing Loop

Following the boot pass, DSIM begins the simulation process proper. The simulation is carried out in a loop which passes repeatedly through the following two steps:

- All the state change events for the current time are read off a queue and applied to the relevant nets. This process results in a new set of net states.

- Where a net changes state, all the models with input pins attached to the net are re-simulated. Where their outputs change state, this creates new events which are placed on the event queue.

Of course, different models will create events which fall due for processing at different times. The DSIM kernel thus has to order all the new events created at the end of each cycle round the loop.

It is also worth pointing out that our scheme quite happily supports models which have a zero time delay. In this context, events generated with the same time-code are processed in batches (one batch equals one trip round the loop), according to how they were generated.

Termination Conditions

Simulation stops when one of the following occurs:

- The specified stop time is reached.
- The event queue becomes empty. This means that the circuit has reached a permanently stable condition.
- A logic paradox with zero time delay occurs such that the current time ceases to advance, despite repeated cycles round the event processing loop.
- A system error such as running out of event queue memory arises. This is unlikely to occur in normal use unless there is something unstable about your design, perhaps leading to a high frequency (e.g. 100MHz) oscillation somewhere.

Using Digital Graphs




To perform a transient analysis of a digital circuit:

1. Add generators as necessary around the circuit, to drive the circuit inputs.
2. Place probes around the circuit at points of interest. These may be at points within the circuit, as well as obvious outputs.
3. Place a *Digital* graph.
4. Add the probes (and generators, if desired) to the graph.

Only *Digital* generators should be used to generate digital signals - note in particular that the *Pulse* generator is intended for generating ***analogue*** pulses.

Only voltage probes should be used for probing digital nets - using current probes will force PROSPICE to perform a mixed mode simulation.

5. Edit the graph and set up the simulation *Stop Time* required, as well as labelling the graph and setting control properties if required.

6. Invoke the *Simulate* command on the *Graph* menu against the graph, or press the space-bar.
-  See THE SIMULATION PROCESS on page 63 for general information on how to simulate a graph.
 -  See Placing Generators on page 94 for information on placing generators and Placing Probes on page 101 for information on placing probes.
 -  See Adding Traces To A Graph on page 60 for more information about adding probes to graphs.

How Digital Data is Displayed

Normal Traces

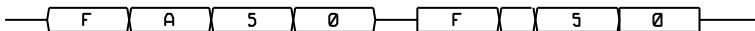
All output values from the DSIM simulator engine is in terms of six digital states:

State Type	Keyword	Graphical Appearance
Strong High	SHI	High level in cyan.
Weak High	WHI	High level in blue.
Floating	FLT	Mid level in white.
Contention	CON	Mid level in yellow.
Weak Low	WLO	Low level in blue.
Strong Low	SLO	Low level in cyan.

You can thus determine the state of a trace at a particular time, either by looking at the colour and position of the trace line, or by maximising the graph, positioning a cursor over the desired position, and reading the state off the status line. The logic levels at the cursor position are also displayed just to the right of the trace labels.

Bus Traces

Bus traces (resulting from the placement of voltage probes on bus wires) display the hex values of the bus bits between cross over lines:



Where one or more of the bus bits is neither high nor low, the bus line is drawn at the mid level. Also, if the edge transitions are too close together to display the hex value, then it is omitted. The value can still be established by positioning a cursor over it.

MIXED MODE TRANSIENT ANALYSIS

Overview

The mixed mode graph allows both digital traces and analogue waveforms to be displayed above the same x-axis which represents the advance of time. Mixed mode analysis is, in fact, used whenever a circuit contains both analogue and digital models, but the mixed mode graph is the only type capable of displaying both types of waveform simultaneously.

Method of Computation

Mixed mode transient analysis combines both the *Non-Linear Nodal Analysis* of SPICE3F5 with the *Event Driven Simulation* of DSIM. The detailed implementation of this is extremely complex, but the basic algorithm may be summarized as follows:

- Prior to simulation, each net is analysed to see whether analogue, digital or both types of model connect to it. Where analogue voltages drive digital inputs, PROSPICE inserts ADC interface objects, and where digital outputs drive analogue components, it inserts DAC interface objects.

Where several digital inputs are driven from the same net, multiple ADC objects are created so that different logic switching levels and loading characteristic can be modelled for each input. Similarly, in the rarer case of several digital outputs being connected together, multiple DAC objects are created.

- The operating point is established as described below.
- Simulation then proceeds as for a normal analogue analysis except that the ADC generate a digital event whenever their input voltages cross the switching threshold. As soon as this happens, the analogue simulation is suspended, and digital simulation is carried to establish the effect of these new events.
- When digital simulation results in a change of state for a DAC interface object, the analogue simulation is forced to simulate carefully around the switching time. In fact, the DAC outputs model a transition time (rise or fall) and a number of timepoints will be simulated during this period

Finding the Operating Point

Establishing the operating point is especially tortuous for mixed mode simulation, because the state of the analogue circuit affects the state of the digital components and vice versa. Essentially, PROSPICE proceeds as follows:

- Initial condition (IC) values are applied to both analogue and digital nets; other nets are assumed to start at zero volts.

- The nodal matrices are then constructed and solved as per normal SPICE simulation.
- For each such iteration, the digital circuit is re-processed to allow for changes in the input to the ADCs. Where such changes propagate through several digital models, the digital circuit is allowed to iterate to stability.
- DAC objects re-assign their outputs according to any changes of state in the digital circuit. If the digital circuit does not settle, this is ignored as it may be a transient citation.
- Iteration round this loop continues until a steady state is found, or until the iteration limit is reached.

Using Mixed Graphs

Mixed graphs are used in exactly the same way as *Analogue* graphs, except that you can add digital traces to them as well. To add the first digital trace to a mixed graph you should use the *Add Trace* dialogue form. Thereafter, dragging a probe on to the analogue part of the graph creates a new analogue trace, whilst dragging it onto the digital part creates a new digital trace.

FREQUENCY ANALYSIS

Overview

With *Frequency Analysis* you can see how the circuit behaves at different frequencies. Only one frequency is considered at a time, so the plots are *not* like those seen on a spectrum analyser, where all frequencies are considered together. Instead, the simulation is similar in effect to connecting a frequency generator to the input, and looking at the output with an AC voltmeter. As well as the *magnitude*, however, the *phase* of the probed signal can also be seen.

Frequency analysis produces frequency response or *Bode* plots. It is useful in checking that filters are behaving as expected, or that amplifier stages will work correctly across the required frequency range.

Frequency graphs can also be used to plot small signal input and output impedance against frequency.

Method of computation

Frequency analysis is performed by first finding the operating point of the circuit, and then replacing all the active components with linear models. The internal capacitances of the active devices are calculated at the operating point, and assumed not to vary much over the working voltage swings of the circuit. All the generators, apart from frequency reference generators

(see below) are replaced by their internal resistances. This causes power lines to be effectively connected together, as is normally the case during frequency computations.

The analysis is then performed with complex numbers in a linear fashion. The frequency of interest is gradually increased from the starting to the concluding frequency in even increments. The linear nature of this analysis makes it typically much faster than transient analysis, even though complex numbers are used.

It is important to remember that frequency analysis assumes a linear circuit. This means that a pure sine wave at the input will produce a pure sine wave at the output, over all the frequencies probed. Of course, no real active circuit is purely linear but many are close enough to allow this form of analysis. There are also circuits which are not at all linear, such as line repeaters with schmitt trigger inputs. For non-linear circuits the term 'frequency response' has no real mathematical meaning, and so any frequency simulation will not produce meaningful results. Should you be interested in the frequency domain behaviour of such circuits, then *Fourier* analysis is much more appropriate.

Using Frequency Graphs

To calculate the magnitude of sine waves at the *output*, we must inject a reference sine wave at the *input*. PROSPICE will do this automatically, but needs to know where the input of the circuit is. To do this, you must give each frequency graph a '*Reference Generator*' to tell the simulator where to inject the reference signal. There are three ways of doing this:

- Use the *Reference* field on the *Edit Frequency Graph* dialogue form to select an input generator. This object can be any ordinary single pinned generator object, or the FREQREF primitive from the ASIMMDLS library.
- Tag and drag any analogue generator onto the frequency graph.
- Use the *Add Trace* dialogue form to add a generator as a **REF**.

The FREQREF primitive is useful for defining models that drive the circuit - a microphone model for example. This could contain an explicit named generator, used as a reference when the effects of the rest of the microphone model (such as modelled frequency response) need to be taken into consideration.

The second and third techniques will be used more often. The action of dragging a generator onto a frequency graph is different to dragging it onto a transient graph. Instead of adding the generator's probe, the generator becomes the circuit reference point. The generator need not be a *Sine* generator - *DC*, *Pulse* and *Pwlin* will all work as well. If you want to add the generator as a probe, it can still be done using the *Add Trace* dialogue form.

The magnitude of the reference is always 1 volt, the phase is always 0 degrees. The internal resistance of the reference generator will follow whatever was defined for the generator in the

first place. This is used for the dB calculations, i.e. $0\text{dB} = 1$ volt. ISIS will limit very small values, to avoid $\log(0)$, to -200dB .

Frequency graphs always have both left and right y-axes. The left y-axis is used to display the magnitude of the probed signal, and the right y-axis is used to display the phase. If you drag a probe onto the left of the graph, it will display magnitude - if you drag it onto the right, the phase. The x-axis is used to display the frequency of the reference generator. A logarithmic scale is always used for the frequency. The left y-axis may be displayed either in dBs or normal units, and the right y-axis is always in degrees.

To see the frequency response of a circuit:

1. Place probes around the circuit at points of interest.
2. Place a *Frequency* graph.
3. Add the probes to it. Add to the left for magnitude, and to the right for phase. You may well want to add probes twice in order to see both phase and magnitude.
4. Edit the graph (point at it and press CTRL+'E') and set the required start and stop frequencies, and any *Simulation Control Properties* required.
5. Press the space bar to invoke PROSPICE.

The samples files ZIN.DSN and ZOUT.DSN show how to combine frequency graphs with trace expressions to plot input and output impedance.

DC SWEEP ANALYSIS

Overview

With a DC Sweep analysis you can see how *changing* a circuit will affect its operation. This is achieved using *Property Expression Evaluation* in the PROSPICE simulator engine. The sweep graph defines a variable that can be swept in even steps over a user-defined range. The sweep variable can appear in any property within the circuit, such as a resistor value, a transistor gain or even the circuit temperature.

A *DC Sweep* curve shows the steady state voltage (or current) levels of the probed points around the circuit, as the swept variable is increased. It can be used to plot the DC transfer characteristic of a circuit by assigning the swept variable to a generator value, as well as plotting the effect of changing component values on the operating point of the circuit.

Method of computation

In a DC sweep PROSPICE will repeatedly find the operating point of the circuit, incrementing the swept variable between calculations. PROSPICE will re-calculate all the variables used

between steps, so the swept variable can be used as often as is required, and variables based on the swept variable can also be used. Any circuit initialisation parameters will be honoured for every operating point calculated.

Using DC Sweep Graphs

In common with analogue transient analysis (see *Analogue Transient Analysis* on page 68), either the left, right or both y-axes may be used. The graph x-axis shows the swept variable. Trace expressions, described in *The Add Trace Command Dialogue Form* on page 62, may be used in DC sweeps. When choosing the number of steps, bear in mind that simulation time is roughly proportional to the number of steps used.

To plot the transfer function of a circuit:

1. Place a *DC* generator on the input of the circuit. Set its value to be X.
2. Place one or more probes at appropriate outputs of the circuit.
3. Place a *DC Sweep* graph, and add the probes to it - see *Adding Traces To A Graph* on page 60.
4. Edit the graph (CTRL+'E') and set the start and stop values to the extremes of the input sweep that you want. Check that the swept variable is X.
5. Press the space bar to invoke PROSPICE.
6. If the resulting traces look disjointed or angular when you zoom in on the graph, increase the number of steps in the *Edit DC Sweep Graph* dialogue form.

To see the effect of altering circuit values:

1. Set up the circuit as you would for a transient analysis. Add probes and generators at appropriate points in the circuit.
2. Edit the components whose values you are interested in. Set their values to be expressions containing X, the swept variable. You can edit just one, or several components.
3. Place a *DC Sweep* graph, and add the probes and generators to it - see *Adding Traces To A Graph* on page 60.
4. Edit the graph, and set the sweep parameters up accordingly.
5. Press the space bar to invoke PROSPICE.

AC SWEEP ANALYSIS

Overview

This type of analysis creates a family of frequency response curves for a different values of a swept variable.

The main use for this graph type is in seeing how a particular component value affects the frequency response of your circuit.

Method of Computation

This analysis is computed exactly as an ordinary frequency response except that multiple runs are performed, one run for each value of the sweep variable.

The restriction of a linear circuit applies to this analysis as it does for a normal frequency analysis - see Frequency Analysis on page 75.

Using AC Sweep Graphs

As with *Frequency* analysis, the left and right axes display gain and phase respectively. Also, an input generator must be specified as a reference point for gain calculations.

To see the effect of altering a circuit parameter on frequency response:

1. Set up the circuit as you would for a frequency analysis.
2. Edit the components of interest. Set their properties to be expressions containing **X**, the swept variable. You can edit just one, or several components.
3. Place an *AC Sweep* graph, and add the probes to it - see Adding Traces To A Graph on page 60.
4. If you do not have a generator on the input already, then place one.
5. Tag and drag a generator on the input of the circuit onto the graph to be the reference generator.
6. Edit the graph (CTRL+'E') and set the sweep parameters accordingly. Set the *Freq* parameter to the frequency of interest.
7. Press the space bar to invoke PROSPICE.

DC TRANSFER CURVE ANALYSIS

Overview

This graph type is specifically designed for producing characteristic curve families for semiconductor devices, although it occasionally finds other applications. Each curve consists of a plot of operating point voltage or current against the value of a nominated input generator which is swept from one DC value to another. An additional generator may also be stepped to produce a set of curves.

Method of Computation

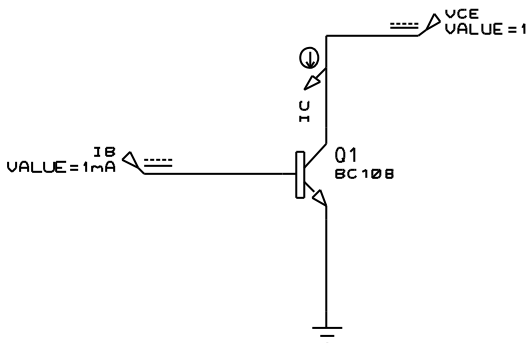
This is very similar to the computation for a DC Sweep analysis except that two values can be swept. The operating point is found for the starting value of each generator and the first generator is then stepped through the specified range. After each sweep of the first generator, the second generator's value is incremented; a new curve is plotted for each discrete value of the second generator.

Using Transfer Graphs

In common with analogue transient analysis (see Analogue Transient Analysis on page 68), either the left, right or both y-axes may be used. The graph x-axis shows the first swept variable, and a separate curve results for each value of the second generator. Trace expressions, described in *The Add Trace Command Dialogue Form* on page 62, may also be used as required.

To plot the transfer curves for an transistor:

1. Set up a circuit similar to the one below:



I_B is a current source and V_{CE} is a voltage source. I_C measures collector current.

2. Place a *Transfer* graph, and add the probe IC to it.
3. Point at the graph and press CTRL+E to edit it.
4. Assign *Source 1* to VCE and *Source 2* to IB.
5. Set the voltage and current ranges to something sensible for the transistor concerned - e.g. 0 -> 10V for VCE and 100uA -> 1mA for IB.
6. Adjust the number of steps for IB to give sensible intermediate values. Bear in mind that their will be one more curve than the number of steps, because one step implies two discrete values.
7. Close the dialogue form and select OK to simulate the graph.

NOISE ANALYSIS

Overview

The SPICE simulator engine can model the thermal noise generated in resistors and semiconductor devices. The individual contributions of noise are summed together at a *voltage* probe in the circuit for a range of frequencies. The noise voltage, normalised with respect to the square root of the noise bandwidth, can then be plotted against frequency. Traces on a noise graph always have units of $V/\sqrt{\text{Hz}}$.

Two types of noise figure are computed - *Output Noise* and *Equivalent Input Noise*. The computation of the latter enables comparison of the noise with the input signal, or input noise, since it represents the level of noise at the input that would be required to create the actually noise at the output taking into account the circuit gain at a particular frequency.

Placing a probe on the left axis displays output noise, whilst equivalent input noise can be displayed by dragging the probe onto the right axis.

Noise analysis tends to produce extremely small values (of the order of nanovolts). For this reason, there is the option to display them in dBs. The 0dB reference is $1V/\sqrt{\text{Hz}}$.

Correct modelling of noise for circuits involving IC macro models is not guaranteed in any way, because these models may well use linear controlled sources to model basic device behaviour only.

Method of calculation

The operating point of the circuit is computed in the normal way, and then the circuit model is modified to correctly sum the noise contributions. All generators except the input reference are ignored in noise analysis (except during computation of the operating point) and so do not have to be removed before the analysis. The PROSPICE engine will compute the thermal

noise for each voltage probe in the circuit, including those associated with generators and tapes, as it has no way of knowing which probes you wish to look at after the analysis. Noise currents are not supported, and the PROSPICE engine will ignore any current probes. A separate simulation is done for each probe in a noise analysis, so the simulation time is roughly proportional to the number of voltage probes placed.

Using Noise Graphs

When using noise analysis, it is important to remember that the effects of noise pick-up from external electric or magnetic fields are not modelled. In many situations where noise is critical the noise picked up in the input leads will dominate the noise performance of the system.

If you have partitioned the circuit using tapes (see TAPES AND PARTITIONING on page 122), you must be aware that only the current partition is considered by the PROSPICE simulator engine. A noise figure taken in isolation is still useful, but the tapes must be removed to see the equivalent noise of the entire circuit.

In order to determine the source of noise in the circuit, the individual noise voltages are entered in to the simulation log. Each probe is considered in turn, and the noise contribution for each component given. The contributions are computed, and printed, as squared values. This process is done at the starting and ending frequencies. If you have a large number of probes, there may well be a great deal of result data printed.

To analyse the noise in a circuit:

1. Place a *Noise* graph, and edit it to set the frequency range of interest, and the input reference generator.
2. Add the voltage probes at the circuit outputs, or other points of interest, to the graph. see Adding Traces To A Graph on page 60.
3. Press the space bar to invoke the PROSPICE simulator engine.
4. If the noise level needs to be reduced, examine the simulation log (CTRL+'V') to determine the source of the noise.

DISTORTION ANALYSIS

Overview

Distortion analysis determines the level of distortion harmonics produced by the circuit under test. These can be either the 2nd and 3rd harmonics of a single fundamental or the intermodulation products of two test signals.

Distortion is created by non-linearities in a circuit's transfer function - a circuit comprising linear components only (resistors, capacitors, inductors, linear controlled sources) will not

produce any distortion. SPICE distortion analysis models distortion for diodes, bipolar transistors, JFETs and MOSFETs.

Correct modelling of distortion for circuits involving IC macro models is not guaranteed in any way, because these models may well use linear controlled sources to model basic device behaviour only.

Similar information can be established using a *Fourier* analysis, but the *Distortion* analysis is also able to show how the distortion varies as the fundamental frequency is swept.

Method of Computation

This analysis is based on the small signal (AC) models for the devices in the circuit so the first step is to compute the operating point. Each non-linear device model then contributes a complex distortion value for the appropriate harmonics, dependent on how much the device is seeing of the input fundamental. The extent to which these harmonics appear at the output determines the values plotted. The process is repeated across the specified range of input frequencies. In fact, the mathematics of this analysis is extremely complicated and involves the construction and manipulation of Taylor series to represent the device non-linearities.

Note that complex values are used, so the analysis yields information about both the magnitude and phase of each harmonic.

For single frequency harmonic distortion, two curves are produced for each trace on the graph - one for 2nd harmonic and one for 3rd harmonic.

For intermodulation distortion, two input frequencies are used, specified in terms of a ratio between the 2nd frequency (F2) and the fundamental (F1). Three curves per trace are displayed showing the intermodulation artefacts at F1+F2, F1-F2 and 2F1-F2.

Some care is needed in choosing the ratio F2/F1 since mathematically oddities can otherwise occur. For example, if F2/F1 is 0.5, the value of F1-F2 is F2 and the value of the F1-F2 plot will then be meaningless as it coincides with the second fundamental. In general you should try to choose irrational numbers for this ratio. F2/F1=49/100 would be a much better choice.

Note that there is a constraint of $F2/F1 < 1$.

Using Distortion Graphs

Distortion graphs show the magnitude of harmonics on the left axis and the phase (normally of less interest) on the right axis. The test frequency (F1) is plotted on the x-axis.

For Harmonic Distortion analysis (one input frequency F1) two curves are plotted per trace, one each for the artefacts at 2F1 and 3F1.

For Intermodulation Distortion analysis (two input frequencies (F1 and F2) three curves are plotted per trace showing artefacts at F1+F2, F1-F2 and 2F1-F2.

In either case, which curve is which may be established by placing a cursor on the graph - the curve you are pointing at will be identified on the right hand side of the status bar.

FOURIER ANALYSIS

Overview

Fourier Analysis is the process of transforming time domain data into the frequency domain and the result is similar to that obtained by connecting up a spectrum analyser instead of an oscilloscope. It is especially useful in analysing the harmonic content of signals, perhaps to look for particular types of distortion but has many other uses also.

Method of Calculation

Fourier analyses are created by first executing a *Transient Analysis* and then performing a Fast Fourier Transform on the resulting data. This process involves discrete time sampling of the time domain data with the result that the use Nyquist Sampling criterion applies. Put simply, this means that the highest frequency that can be observed is half that of the sampling frequency. However, other misleading effects can occur due to *aliasing* of the sampling frequency with harmonics of the input signal that are higher than half the sampling frequency. To minimize these effects, various types of *window* can be applied to the input data prior to the FFT.

Using Fourier Graphs

In the first instance, you need to set up your circuit as for a *Transient Analysis* except that you should use a *Fourier* graph instead of (or as well as!) an *Analogue* graph.

To analyse the frequency content of a signal:

1. Set up the circuit as for a *Transient Analysis*.
2. Add a *Fourier* graph to the design and drag probes connected to the points of interest onto it.
3. Choose start and stop times and frequency/resolution values to suit the signal you are analysing. If possible, choose a time interval and frequency resolution that correspond to the fundamental frequency of the signal being analysed.
4. Press the space bar to invoke PROSPICE.

If the start and stop times are the same for both Transient and Fourier graphs, PROSPICE will not need to resimulate the circuit between transient and fourier analyses. Instead, ISIS will just perform an FFT on the existing time domain data.

AUDIO ANALYSIS

Overview

PROTEUS VSM incorporates a number of features that enable you to *hear* the output from your circuits (providing you have a sound card, of course!). The major component of this is the *Audio* graph. This is essentially the same as an *Analogue* graph except that after simulation, a Windows WAV file is generated from the time domain data and played through your sound card.

The WAV files can also be exported for use in other applications.

Method of Calculation

Audio analyses are performed in exactly the same way as for *Transient Analysis* except that after simulation, the data is re-sampled at one of the standard PC sampling rates (11025, 22050 or 44100Hz) then written out in WAV format using the standard Windows functions provided for this purpose. Finally the command is sent to play the WAV file to your audio hardware.

Using Audio Graphs

In the first instance, you need to set up your circuit as for a *Transient Analysis* except that you should use an *Audio* graph instead of an *Analogue* graph.

To listen to the audio output of a circuit:

1. Set up the circuit as for a *Transient Analysis*.
2. Add an Audio graph to the design and drag a probe from the output of the circuit onto it.
3. Choose start, stop times and loop times to generate a waveform of reasonable length with the minimum of actual simulation. Creating one second of audio by analysing 1ms and looping it 1000 times is dramatically faster than analysing the circuit for the whole 1s.
4. Choose a sample resolution and rate to suit the nature of the signal you want to hear. Use 16 bit resolution unless you are creating big files and/or are short of disk space. Most PC sound speaker systems will not benefit much from 44.1KHz sampling.
5. Press the space bar to invoke PROSPICE.
6. Press CTRL-SPACE to replay the audio, without resimulating, if required.

INTERACTIVE ANALYSIS

Overview

The Interactive Analysis type combines the advantages of interactive and graph based simulation. The simulator is started in interactive mode, but the results are recorded and displayed on a graph as with Transient Analysis.

This analysis is especially useful in examining what happens when a particular control is operated in a design, and may be thought of as combining a storage oscilloscope and logic analyser into one device.

Method of Computation

The method of computation is identical to that for a mixed mode transient analysis, except that the simulator is run in interactive mode. Consequently, the operation of switches, keypads and other actuators in the circuit will have an effect on the results. Also, the simulation will proceed at a speed determined by the *Animation Timestep*, rather than at the maximum rate possible.

- Beware that very large amounts of data may be captured. A processor clocking at realistic speeds will generate millions of events per second and these will occupy many megabytes if captured and displayed on a graph – especially if you are probing data or address busses. You can quite easily crash your system if ISIS ends up loading 20 or 30Mb of results data.

You may do better to use the *Logic Analyser* if it is not possible to capture the required data in a relatively short simulation run.

- As with ordinary *Interactive Simulation*, multi-partitioning of the circuit is not supported and any tape objects not set to *Play* mode will be automatically removed from the circuit.

Using Interactive Graphs

The usual scenario for interactive analysis is that you are testing a design with interactive simulation, and find that something odd happens when particular controls are operated. In the first instance, you may try using the virtual instruments to see what is going on, but in some cases there is a need to capture the results to a graph and study them at your leisure.

To perform an interactive analysis:

1. Add probe objects to the points of interest.
2. Place an *Interactive* graph on a free part of the schematic and drag the probes onto it.
3. Edit the graph and choose appropriate start and stop times. Note that PROSPICE will not capture data to the probes prior to the start time – this is to reduce the amount of data that will be pulled into ISIS.
4. Set any interactive controls on the schematic to suitable initial states.
5. Get ready to perform any interactive operations and then press the SPACE bar. You have to operate the active components within the time you chose in step 4 if the effect of these actions is to be recorded. If this proves difficult, you can either increase the stop time, or reduce the *Timestep Per Frame* setting.

DIGITAL CONFORMANCE ANALYSIS

Overview

A conformance analysis compares one set of digital simulation results against another. The idea is that a design that has been previously accepted as working can be quickly re-tested after modification in order to prove that there have been no unwanted side effects from the change. This is particularly relevant in micro-controller based applications where the entire application may need to be re-tested after changes have been made to the firmware code.

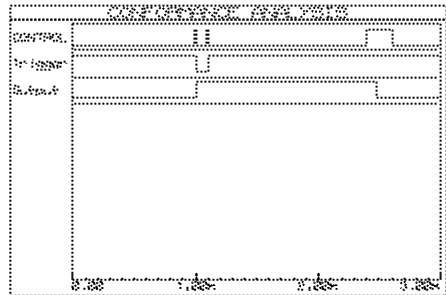
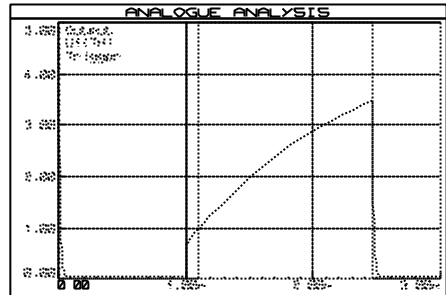
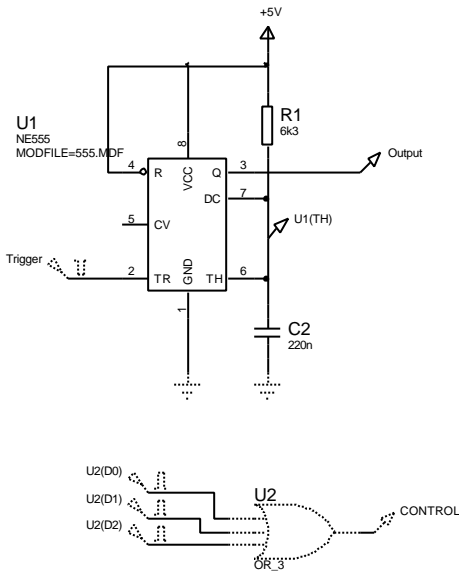
Method of Computation

The method of computation is identical to that for a digital transient analysis, except that two sets of results can be stored in the graph. We call the two sets of results the *test results* and the *reference results*.

Conformance or non-conformance is determined by comparing the test and reference results at each edge of the first trace. We call this trace the *control trace* and it is displayed in a different colour to distinguish it from the others. Very significantly, there is no requirement for the edges in the test and reference copies of the control trace to occur at the same times. This means that changes in the absolute timing of events within the result data do not necessarily imply non-conformance. This is particularly relevant in micro-controller applications where *any* changes to the code will be bound to affect the absolute timing of events within the system. In such cases, the control trace may be generated by the code itself, on entry and/or exit to the software routines under test.

Using Conformance Graphs

Typically, conformance analyses will be used as part of a test strategy, most usually in an embedded systems application although we also use them in-house for testing our simulator models. In the simple example, below, will show how to use a conformance graph to test the operation of 555 monostable.



The circuit under test consists of U1, R1 and C2 which are wired as a classic 555 monostable. The circuit is triggered at 1ms by a digital pulse generator, and the resulting output waveforms are displayed on the analogue analysis graph. To test the correct operation of the circuit, it would be reasonable to establish the following facts:

- That the output is low before the trigger pulse.
- That the output goes high soon after the trigger goes low.
- That the output remains high when the trigger returns high.
- That the output remains high for a period of around 1.5ms.
- That the output returns low after this time.

To achieve this using the conformance graph, we need sample the output data around each edge transition of the trigger signal, and also at either side of the output signal. We can determine a tolerance for the delay time of the monostable by choosing the spacing between the last two sampling points.

This is achieved using a series of digital pulse generators whose outputs are combined using an OR gate. The width setting on each pulse generator determines the timing tolerance for each expected event. For example, the third pulse generator is set to trigger between 2.4ms and 2.6ms, giving timing tolerance of $\pm 100\mu\text{s}$ on the width of the expected output pulse.

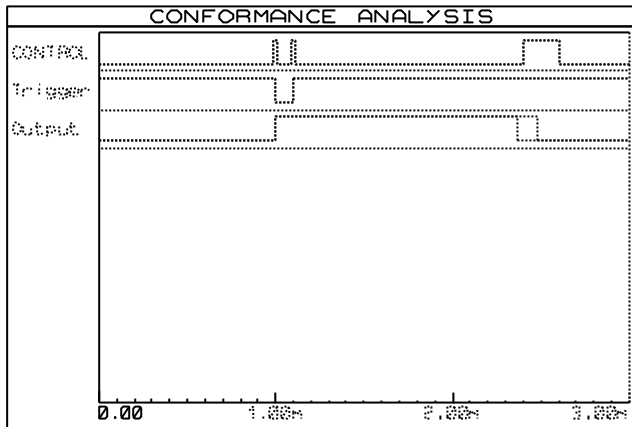
Note that you could have used a digital clock or pattern generator to generate the control signal; the use of a simple clock will result in verification at regular time intervals which will, in fact, suffice for many applications.

The general procedure may be summarized as follows:

To set up a conformance analysis:

1. Decide on what exactly you want to test and at what times.
2. Design a test case schematic which generates the outputs you wish to verify and a control signal that changes state at each time you wish to validate the results.
3. Place a conformance graph and set it up exactly as you would for a *Digital Analysis*, making sure that the control trace is at the top of the graph.
4. Run the simulation and verify that the results are as expected, and that the transitions on the control trace occur at the times that you wish to verify the results.
5. Edit the conformance graph and press the *Store Results* button. This will make the currently displayed results into the reference results. These then are displayed in a dimmed colour and the graph will compare any new results against them when it is re-simulated.
6. Save the design - it is now ready to be used for verification purposes at a future date.

Let us now suppose that due to component shortages, the values of R1 and C2 must be changed. If we make C2 100nf, can we make R1 15k, and will the design still work within spec? The results of this experiment are shown below:



The answer is no. The output pulse is now slightly too short, and the following information appears in the simulation log for the graph:

```
Comparing Results...
Data mismatch at time 2.40m in trace 'Output'.
Data signatures were 'L' and 'H'.
Conformance analysis FAILED.
```

Note the graph cursor is positioned at the time of the first discrepancy i.e. 2.4ms

There are in fact three ways to invoke a conformance analysis:

- By re-simulating the graph in the usual way - e.g. by pressing the space bar or using the *Simulate Graph* command on the *Graph* menu.
- Using the *Conformance Analysis* command on the *Graph* menu. This re-simulates all the conformance graphs in the design and reports any errors.
- For advance users only - from the command line. Entering

```
ISIS <filename> /V
```

will perform a global conformance analysis as above. If any graph fails ISIS sets an error level of 1. This allows multiple test-cases to be automatically validated from a batch file.

DC OPERATING POINT

This is the one analysis type for which there is not a corresponding graph. Instead, the information computed can be viewed through a point and shoot user interface within ISIS.

It is important to appreciate that the operating point is computed for the *Current Graph*, and that there must be a graph set up on the schematic in order to compute the operating point. The reason for this is that without a graph, there is no way for the system to know which parts of the circuit to simulate. In addition, the functionality related to *Property Expression Evaluation* and the ability to set properties on a graph means that actual component values may depend on which graph is used.

To view operating point values:

1. Set up the circuit as for a *Transient Analysis*.
2. Add a graph to the circuit and add probes and generators to it appropriate to the section of circuitry you wish to simulate.
3. Simulate the operating point by pressing ALT+SPACE. Voltage and current probes will display their operating point DC values immediately.
4. Click on the *Multimeter* icon.
5. Point at any component and click left to view its operating point information

A couple of things to note:

- It is possible to compute the operating point without a graph. In this case, an attempt is made to simulate the entire schematic as one partition – i.e irrespective of any probes, generators or tapes. This will fail if there are objects placed which have not been modelled.
- Components which are actually sub-circuit parents, or which are modelled by MDF or SPICE files will display basic operating point information only – i.e. their node voltages.

GENERATORS AND PROBES

GENERATORS

Overview

A *generator* is an object that can be set up to produce a signal at the point at which it is connected. There are many types of generator, each of which generates a different sort of signal:

- *DC* - Constant DC voltage source.
- *Sine* - Sine wave generator with amplitude, frequency and phase control.
- *Pulse* - Analogue pulse generator with amplitude, period and rise/fall time control.
- *Exp* - Exponential pulse generator - produces pulses with the same shape as RC charge/discharge circuits.
- *SFFM* - Single Frequency FM generator - produces waveforms defined by the frequency modulation of one sine wave by another.
- *Pwlin* - Piece-wise linear generator for arbitrarily shaped pulses and signals.
- *File* - As above, but data is taken from an ASCII file.
- *Audio* - uses Windows WAV files as the input waveform. These are especially interesting when combined with Audio graphs as you can then listen to the effect of your circuit on audio signals.
- *DState* - Steady state logic level
- *DEdge* - Single logic level transition or edge.
- *DPulse* - Single digital clock pulse.
- *DClock* - Digital clock signal.
- *DPattern* - An arbitrary sequence of logic levels.

The first eight generator types are considered to be analogue components and are simulated by the SPICE simulator kernel whilst rest relate to digital circuitry and are handled by DSIM.


Placing Generators

To Place a Generator:

1. Obtain a list of generator types by selecting the *Generator* icon. The list of generator types is then displayed in the *Object Selector*.
2. Select the type of generator you wish to place from the selector. ISIS will show a preview of the generator in the *Overview Window*.
3. Use the *Rotation* and *Mirror* icons to orient the generator according to how you want to place it.
4. Place the mouse in the *Editing Window*, press down the left mouse button and drag the generator to the correct position. Release the mouse button.

You can place a generator directly on to an existing wire by placing it such that its connection point touches the wire. Alternatively you can place several generators in a free area of your design, and wire to them later.

When a generator is placed unconnected to any existing wires, it is given a default name of a question mark (?) to indicate that it is unannotated. When the generator is connected to a net (possibly when placing it if it is placed directly onto an existing wire) it is assigned the name of the net, or, if the net itself is unannotated, the component reference and pin name of the first pin connected to the net. The name of the generator will automatically be updated as it is unwired or as it is dragged from one net to another. You can assign your own name to the generator by editing the object, in which case the name becomes permanent and is not updated.

 See *NET NAMES* in the ISIS manual for an explanation of nets and net-naming.

Editing Generators

Any generator may be edited using any of the general ISIS editing techniques, the easiest of which are either to click right (to tag) and then left (to edit), or to point at the generator and press CTRL+E.

The *Edit Generator* dialogue form provides a number of common fields and then a further set of fields that change according to the generator's type. The common fields are explained below:

Name

The name of the generator.

You can change the name of a generator by typing in a new name into the name field. Once you have manually changed the name of a generator, ISIS will never replace it, even if you move the generator to a new net.

	If you want a probe to revert to automatic naming, clear its name to the blank string by pressing ESC once, and then clicking OK.
Type	The type of generator. You can change the type of generator from the type placed by selecting the button for the new type required.
	This control determines the generator specific fields that are displayed on the right hand side of the dialogue.
Current Source	With the exception of the DIGITAL generator, all the other types are capable of operating either as voltage or current sources. Checking this box causes the generator to be a current source.
Isolate Before	This check box controls whether or not a generator placed in the middle of a wire acts to 'break' the wire to which it is connected, isolating the net to which the generator points from the net behind it. The check box has no affect on a generator connected directly to a net by a single wire.
Manual Edits	When checked, the generator's properties are displayed as a textual property list. This is provided mainly for backward compatibility with previous versions of the software. However, advanced users may wish to use property expressions within generators' properties, and this is only possible in the manual edit mode.

DC Generators

The *DC* generator is used to generate a constant analogue DC voltage or current level. The generator has a single property which specifies the output level.

Sine Generators

The *Sine* generator is used to produce continuous sinusoidal waves at a fixed frequency. A number of parameters can be adjusted:

- The output level is specified as a peak amplitude (VA) with optional DC offset (VO). The amplitude may also be specified in term of RMS or peak-peak values.
- The frequency of oscillation can be given in Hz (FREQ), or as a period (PER), or in terms of the number of cycles over the entire graph.

- A phase shift can be specified in either degrees (PHASE) or as a time delay (TD). In the latter case, oscillation does not start until the specified time.
- Exponential decay of the waveform after the start of oscillations is specified by the damping factor, THETA.

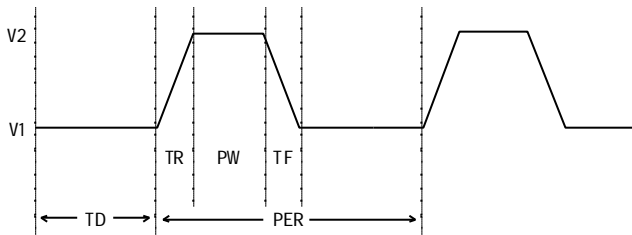
Mathematically, the output is given by:

$$V = VO + VAe^{-(t-TD)THETA} \sin(2\pi FREQ(t + TD))$$

for $t \geq TD$. For $t < TD$ the output is simply the offset voltage, VO.

Pulse Generators

The *Pulse* generator is used to produce a variety of repetitive input signals for analogue analyses. Square, saw-tooth and triangular waveforms can be produced, as well as single short pulses.. Note that the rise and fall times cannot be zero, so a truly square wave is not allowed. This is because instantaneous changes in general are not allowed in PROSPICE. The operation of the pulse generator is best described by the following diagram:



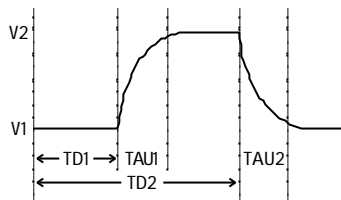
where

- | | |
|-------------|---|
| PER | - The period of the waveform. If none is specified, then FREQ is used instead. |
| FREQ | - The frequency of the waveform. This defaults to one period for a transient analysis. |
| V1 | - The low-level value of the output. |
| V2 | - The high-level of the output. |
| PW | - The time for which the output is at V2 in each cycle. This does not include TR and TF . |
| TR | - The rise time - time taken between LOW and HIGH in each cycle. This defaults to 1ns. |
| TF | - The fall time - time taken between HIGH and LOW in each cycle. This defaults to TR . |

- TD** - The delay time. The output of the generator starts at **V1**, and will remain at this level for **TD** seconds.

Exponential Generators

The *Exp* generator produces the waveforms as an RC circuit under charging and discharging conditions. The parameters are best described by the diagram overleaf.



where the parameters are

- V1** - The low-level value of the output.
- V2** - The high-level of the output.
- TD1** - The start time of the rise curve.
- TAU1** - The time constant of the rise curve. This is the time for the voltage to reach approximately 0.63 of its full potential.
- TD2** - The start time of the fall curve. Note that **TD2** is measured from zero.
- TAU1** - The time constant of the fall curve.

Mathematically, the waveform is defined in three sections:

$$0 \text{ to } TD1 \quad V1$$

$$TD1 \text{ to } TD2 \quad V1 + (V2 - V1) \left(1 - e^{-\frac{(t-TD1)}{TAU1}} \right)$$

$$TD2 \text{ to } TSTOP \quad V1 + (V2 - V1) \left(1 - e^{-\frac{(t-TD1)}{TAU1}} \right) + (V1 - V2) \left(1 - e^{-\frac{(t-TD2)}{TAU2}} \right)$$

Single Frequency FM Generators

The *SFFM* generator produces a waveform that represents the result of frequency modulating one sine-wave with another. Mathematically, this is represented as:

$$V = VO + VA \sin(2\pi F C t + MDI \sin(2\pi F S t))$$

where the parameters are

VO	- The DC offset voltage.
VA	- The carrier amplitude.
FC	- The carrier frequency.
FS	- The signal frequency.
MDI	- The modulation index.

Piece-wise Linear Generators

The *Piece-wise Linear* generator is used to produce analogue signals which are too complicated for a pulse generator, or to re-produce a measured waveform. The output waveform is described using pairs of values for time and output magnitude. The output of the generator at intermediate times is then linearly interpolated between the given times.

The piece-wise linear generator dialogue form consists of a graph editor onto which you can drag and drop data points. Operation can be summarized as follows:

- To place a new data point click left.
- To move a data point, drag it using the left mouse button.
- To delete a data point click right.

The following constraints apply:

- There must always be a point at time zero, although its y-value can be changed - i.e. you can only drag it up or down.
- If you attempt create a vertical edge, the graph editor will separate the two data points by the minimum rise/fall time value.

If you have tabular data, you may find it easier to use manual edit mode. In this case, each data point is specified by a property of the form $V(t)$. The following example shows how this works.

$V(0) = 0$
 $V(2n) = 0$
 $V(3n) = 1$
 $V(5n) = 1$
 $V(6n) = 0$

If you have a large amount of data, you may find it easier to use a FILE generator instead.

File Generators

The *File* generator is used to drive a circuit from an analogue signal that is specified by series of time points and data values contained in an ASCII file. It is thus very similar to the piecewise linear generator except that the data values are held externally rather than being given as device properties.

The dialogue form has only one field, which specifies the name of the data file. There is no default extension for these files, and the file should be located in the same directory as the design file unless a full path is specified.

Data File Format

The ASCII data file should be formatted with one time/voltage pair on each line separated by white space (spaces or tabs, not commas). The time values must be in ascending order, and all values must be simple floating point numbers (no suffixes allowed).

Example

The following example data file produces three cycles of a saw-tooth waveform with rise time 0.9ms, fall time 0.1ms and amplitude 1V.

```
0          0
9E-4      1
1E-3      0
1.9E-3    1
2E-3      0
2.9E-3    1
3E-3      0
```

Audio Generators

The *Audio* generator is used to drive a circuit from a Windows WAV file. In conjunction with *Audio* graphs they make it possible to hear the results of processing a sound signal through a simulated circuit.

- The filename is assumed to have a default extension of WAV, and should be located in the same directory as the DSN file unless a full path is specified.
- The amplitude can be specified either in terms of the maximum absolute value for positive and negative excursions, or as a peak to peak value.
- A DC offset can be specified; if this is zero the output voltage will oscillate about zero.
- For stereo WAV files, you can select which channel is played, or whether to treat the data as mono.

Digital Generators

There are five sub-types of digital generator:

- Single Edge – a single transition from low to high or high to low.
- Single Pulse – a pair of transitions in opposite directions which together form a positive or negative pulse waveform. You can specify either the times of each edge (start time and stop time) or the start time and the pulse width.
- Clock – a continuous pulse train of even mark-space ratio. You can specify the starting value and the time of the first edge as well as the period or frequency. The period specifies the time for a whole cycle, not the width of a single mark or space.
- Pattern – this is the most flexible and can, in fact, generate any of the other types. The pattern generator is defined in terms of the following parameters:

Initial State This is the value at time zero, and also the value used when finding the operating point in a mixed mode circuit.

First Edge This is the time when the pattern actually starts; the output will remain at the initial state value until this time is reached.

Timing Each step of the pattern can take the same time (*Equal mark/space timing*) or can have a different time for high and low values. In this case, the *Pulse* width specifies the time for logic '1' values and the *Space Time* specifies the time for logic '0' values.

Transitions The output can be defined to run continuously till the end of the simulations, with the pattern repeating, or for a fixed number of edge transitions only.

Bit Pattern The default pattern is a simple alternative high-low sequence. Alternatively, a pattern string may be specified. The pattern string can contain the following characters:

- | | |
|------------|--|
| 0,L | - Output waveform is to go to a strong low level (note upper-case 'L'). |
| 1,H | - Output waveform is to go to a strong high level (note upper-case 'H'). |
| l | - Output waveform is to go to a weak low level (note lower-case 'l'). |
| h | - Output waveform is to go to a weak high level (note lower-case 'h'). |
| F,f | - Output is to go to a floating level. |

- Script – the generator will be controlled from a DIGITAL BASIC script. The generator is accessed in the script by declaring a **PIN** variable with the same name as the generator's reference.

PROBES

Overview

A *probe* is an object that can be set up to record the state of the net to which it is connected. There are two types of probe:

- Voltage probes - these can be used in both analogue or digital simulations. In the former they record true voltage levels, whilst in the latter they record logic levels and their strengths.
- Current probes - these can only be used in analogue simulations, and must be placed on a wire such the wire passes through the probe. The direction of measurement is indicated by the current probe graphic.

You cannot use current probes in digital simulations, or on buses.

Probes are most normally used in *Graph Based Simulations*, but can also be used in interactive simulations to display operating point data and to partition the circuit.

Placing Probes

To Place a Probe:

1. First select either the *Voltage Probe* or *Current Probe* icon. ISIS will show a preview of the probe in the *Overview Window*.
2. Use the *Rotation* and *Mirror* icons to orient the probe according to how you want to place it.

Note that, for a *Current Probe* correct orientation is important. Current flow is measured in the direction indicated by the arrow enclosed in a circle which forms part of the probes symbol.

3. Place the mouse in the *Editing Window*, press down the left mouse button and drag the probe to the correct position. Release the mouse button.

You can place a probe directly on to an existing wire by placing it such that its connection point touches the wire. Alternatively you can place several probes in a free area of your design, and wire to them later.

When a probe is placed unconnected to any existing wires, it is given a default name of a question mark (?) to indicate that it is unannotated. When the probe is connected to a net

(possibly when placing it if it is placed directly onto an existing wire) it is assigned the name of the net, or, if the net itself is unannotated, the component reference and pin name of the first pin connected to the net. The name of the probe will automatically be updated as it is unwired or as it is dragged from one net to another. You can assign your own name to the probe by editing the probe in which case the name becomes permanent and is not updated.

Probe Settings

The *Edit Probe* dialogue form allows two parameters to be adjusted:

LOAD Resistance

Voltage probes may be set to impose a load resistance on the schematic. This is useful where there would otherwise be no DC path to ground from the point being measured.

Record Filename

Both current and voltage probes can record data to a file which can then be played back using a *Tape Generator*. This feature allows you to create test waveforms using one circuit and then play them back into another.



See *Tapes and Partitioning* on page 122 for more information.

USING SPICE MODELS

OVERVIEW

Many of the major component manufacturers now provide SPICE compatible simulator models for their product lines. Since PROSPICE is based on the original Berkeley SPICE code, you should have relatively few problems in using such models. However, there are a number of issues which you need to understand before attempting to use 3rd party models:

- A SPICE model is an ASCII netlist file. It contains no graphical information at all and cannot therefore be placed directly on a schematic. This means that you will need an ISIS library part for the device concerned as well as the SPICE model. Sadly, there is no standard file format whatsoever for schematic library parts, so you will usually need to draw this yourself.
- In a SPICE netlist, a model is called up by referring to a subcircuit using an X card. The binding of circuit nodes to model nodes is determined by the order in which circuit nodes are given on the X card. For example

```
XU1 46 43 32
```

means that node 46 of the circuit is connected to node 1 of the model. Similarly node 43 binds to node 2 and node 32 binds to node 3. Unfortunately, this scheme means that the model nodes are not named. Worse still, there is rarely any tie up between the model node numbers and the physical device pin numbers - for one thing, the physical package may have NC (not connected) pins, and these are hardly ever accounted for in numbering the nodes of a SPICE model. There can also be major complications with a multi-element device such as a TL074 - does the model implement one op-amp or four?

In practice, this means that some kind of explicit cross-referencing is needed to tell ISIS which SPICE model node number to use for each pin on the device, since neither the ISIS pin name nor the pin number can be used for the reasons given above. A special device property **SPICEPINS** has been introduced to make this as painless as possible.

- There are many variants of SPICE in existence. The lowest common denominator is SPICE 2 and most models appear to be written to this standard. However, a significant number of manufacturers declare their models to be compatible with PSPICE™, a proprietary version of SPICE 2 which has been developed extensively beyond the original standard. Whilst SPICE3F5 has many features not supported by PSPICE™, PSPICE™ has some primitive types that are different, and also uses a different syntax for some of the newer things that have been added to both products. The implication is, of course, that models specifically written for PSPICE™ may not work with PROSPICE.

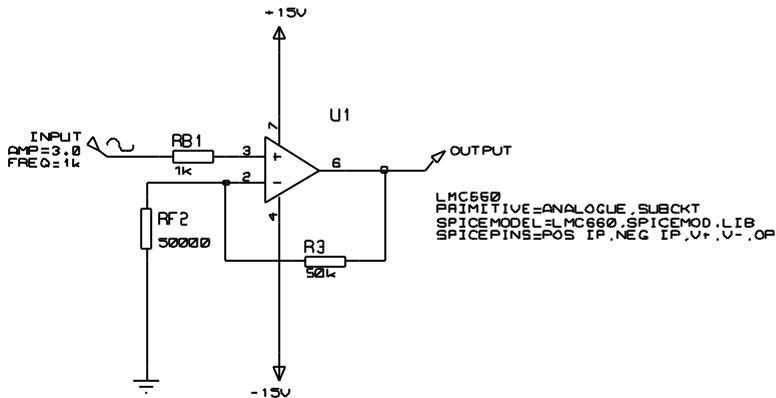
In short, if a 3rd party model does not work properly, the first thing to check is that it was actually written for SPICE 2 or SPICE 3.

The really good news is that we ourselves have gathered together a large number (well over 1500 at the time of writing) of manufacturers' models and created corresponding ISIS library parts for them.

You must, of course, appreciate that we can accept no responsibility for the accuracy or even functionality of these models, and that both ourselves and the manufacturers concerned disclaim any liability for losses of any kind whatsoever arising out of their use. In any case, there is never any guarantee that a circuit simulation will exactly reflect the performance of the real hardware and it is always advisable to build and test a physical prototype before entering into large scale production.

USING A SPICE SUBCIRCUIT (SUBCKT DEFINITION)

Our first example is an LMC660 OPAMP model held in the SPICE file SPICEMOD.LIB. This is provided as the file SPICE1.DSN in the samples directory and is shown below.



The op-amp component has been given three property assignments - **PRIMITIVE**, **SPICEMODEL** and **SPICEPINS**. Taking each of these in turn:

PRIMITIVE=ANALOGUE, SUBCKT

This assignment is the always the same for a component that is modelled by a SPICE subcircuit. It signifies that the component is to be simulated directly by SPICE3F5 and that it is represented by a SPICE subcircuit, rather than being an actual SPICE primitive.

SPICEMODEL=LMC660, SPICEMOD.LIB

This line indicates the name of the subcircuit to use, and the name of the ASCII file that contains the subcircuit definition. Alternatively you could use

```
SPICEMODEL=LMC660
SPICEFILE=SPICEMOD.LIB
```

Some manufacturers supply many models in one file, others supply one file per model. It matters not. The subcircuit name must match exactly that used in the model file, so if the subcircuit were called LMC660/NS you must include exactly this text in the **SPICEMODEL** property.

SPICE model files are searched for in the current directory, and on the *Module Path* as specified on the *Set Paths* dialogue form.

```
SPICEPINS=POS IP,NEG IP,V+,V-,OP
```

The **SPICEPINS** property deals with the issue of binding ISIS pin names to SPICE model node numbers. To understand how this property is used, you need to look at the comments in the original SPICE model file:

```
*////////////////////////////////////
*LMC660AM/AI/C CMOS Quad OP-AMP MACRO-MODEL
*////////////////////////////////////
*
* connections:      non-inverting input
*                   |      inverting input
*                   |      |      positive power supply
*                   |      |      negative power supply
*                   |      |      |      output
*                   |      |      |      |
*                   |      |      |      |
*.SUBCKT LMC660      1      2      99      50      41
*
```

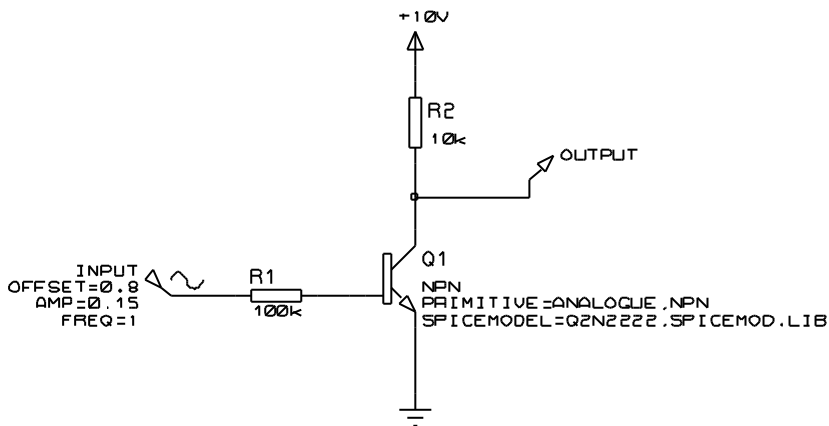
The 'diagram' lists the pin functions and their ordering on the SUBCKT line. It is important to realise that the *numbers* assigned relate to the internals of the model definition and are not in fact relevant to our purpose. The key information is that the non-inverting input is the first node, the inverting input the second and so on. This determines how you construct the value of the **SPICEPINS** property, in that you give the ISIS pin names in the corresponding order. Pin names with spaces are OK, but be careful not to leave spaces before the commas. Alternatively you can put each pin name in quotes - this is essential if you have a pin name that contains a comma.

You need, of course, to know the ISIS pin names which are often hidden. These can be established by moving the mouse over the pin ends of the component, and observing the status line.

USING A SPICE MODEL (MODEL CARD)

For semiconductor devices, specifically diodes, transistors, JFETs and MOSFETs, SPICE models are usually specified in terms of a single MODEL card. This lists the parameter values for the appropriate SPICE primitive. The procedure is similar to that for a SUBCKT model, but actually a little simpler because the issue of pin name translation does not arise. Additionally, ISIS library parts already exist for the various types of SPICE primitive – you will find them in ASIMMDLS.LIB.

Our second example, then, is a 2N2222 transistor, again held in the ASCII file SPICEMOD.LIB. This is provided as the file SPICE2.DSN in the samples directory.



The transistor has been given two property assignments - **PRIMITIVE** and **SPICEMODEL**.

Taking each of these in turn:

PRIMITIVE=ANALOGUE, NPN

This assignment indicates that the device is to be simulated directly by PROSPICE as an NPN type primitive. If you pick a diode, transistor, JFET or MOSFET from the ASIMMDLS library, it will have the appropriate **PRIMITIVE** assignment already in place.

SPICEMODEL=Q2N2222, SPICEMOD.LIB

The `SPICEMODEL` assignment is essentially the same as for a `SUBCKT` model. It specifies the name of the model to use and the SPICE netlist file that contains it. Most manufacturers supply lots of `MODEL` definitions in a single file.

SPICE model files are searched for in the current directory, and on the *Module Path* as specified on the *Set Paths* dialogue form.

SPICE MODEL LIBRARIES

The 3rd party SPICE model libraries supplied with PROSPICE are contained in *binary* library files similar to ISIS device and symbol libraries. We have done this for two reasons:

- Many model files are extremely small and numerous and would waste a tremendous amount of space on hard disks with large cluster sizes if stored individually.
- When many models are contained in a single ASCII file, PROSPICE has to parse the whole file to use just one model and this is relatively slow.

Where a SPICE model is contained in a SPICE model library (SML), the syntax for binding it to an ISIS library part is slightly different. For example, the LMC660 in the first example would instead have the properties:

```
PRIMITIVE=ANALOGUE, SUBCKT           (as before)
SPICEMODEL=LMC660
SPICEPINS=POS IP, NEG IP, V+, V-, OP  (as before)
SPICELIB=NATSEMI
```

The **SPICELIB** property gives the name of the SPICE model library file that contains the part. These files are searched for in the current directory, and on the *Module Path* as specified on the *Set Paths* dialogue form.

SPICE model libraries can be manipulated with the command line tools PUTSPICE.EXE and GETSPICE.EXE although we do not expect that ordinary users to create or gather sufficient numbers of models to warrant their use. Running either program without parameters will give usage information.

***SPICE SCRIPTS**

It is actually possible to type SPICE model definitions directly into ISIS and then call up the models from the relevant components on the schematic. For example, the transistor model for the 2N2222 could be entered into an ISIS script as follows:

```
*SCRIPT SPICE
.MODEL Q2N2222 NPN(IS=3.108E-15 XTI=3 EG=1.11 VAF=131.5 BF=300
NE=1.541
+ISE=190.7E-15 IKF=1.296 XTB=1.5 BR=6.18 NC=2 ISC=0 IKR=0 RC=1
+CJC=14.57E-12 VJC=.75 MJC=.3333 FC=.5 CJE=26.08E-12 VJE=.75
+MJE=.3333 TR=51.35E-9 TF=451E-12 ITF=.1 VTF=10 XTF=2)
*ENDSCRIPT
```

The transistor's **SPICEMODEL** property would then be changed to

SPICEMODEL=Q2N2222

i.e. no filename specified.

This feature might be most useful where a model is obtained on paper, or where you want to experiment interactively with the parameter values.

HANDLING MODEL SIMULATION FAILURES

Subcircuit models vary greatly in complexity and design. As a result, some models are easier to simulate than others. As a general rule, if PROSPICE could not simulate your circuit with its translated model, then not much else could either.

We have often found that problems are greatly worsened by poor circuit design. If you have an application note for the device, then compare it with your circuit to see if, for example, you are not allowing enough input current to bias the input stage successfully. The model designer almost certainly made the model work with circuits similar to those in an application note, rather than in a general case.

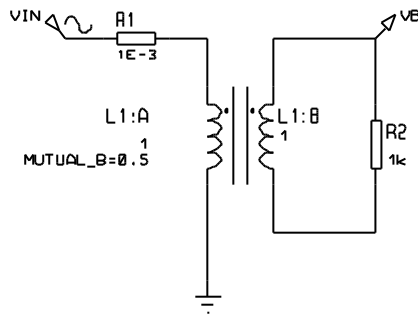
ProSPICE can be made more stable (but less accurate) by increasing the value of **GMIN**. The default is 1E-14, so good values to try are 1E-13 and 1E-12. At values like 1E-6, it is likely that any simulation results will bear no relation to reality at all, so pick the lowest value you can.

ADVANCED TOPICS

GROUND AND POWER RAILS

Why You Need a Ground

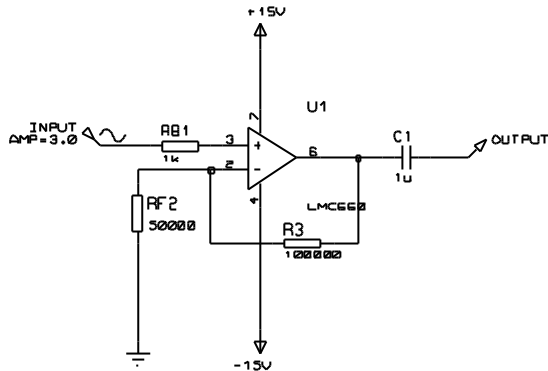
All simulations require a ground net be defined - otherwise, placing a probe on a particular net has no meaning as the voltage on that net must be defined in terms of a fixed reference point. In fact, there is a further requirement that all parts of the circuit must have a DC path to ground, as probing a point in the circuit that is attached to a floating network is equally meaningless. For example, in the circuit below, asking for the voltage at VB is meaningless



because the secondary side is floating. In theory, we could have provided two pinned probes (as with a real multimeter) but there are serious mathematical difficulties in solving circuits without a ground, and more to the point, Berkeley University have not addressed them in SPICE3F5.

The key point, then, is that all sections of your circuit must have a DC path to ground. The good news is that ProSPICE checks this for you, and will report as warnings any nets which it considers do not meet this criterion. In most cases, the simulation will then fail.

Another circuit configuration which has caused trouble in the past is shown overleaf:



Here, the presence of coupling capacitor C1 means that the output probe has no DC path to ground. Consequently, the simulator cannot resolve the operating point for the output, because the operating point is computed with all capacitors open circuit. We have chosen to resolve this by making capacitors very slightly leaky. Therefore, in the absence of any other DC path, the operating point at the output is computed with C1 fully discharged.

The leakyness of capacitors is determined by the simulator control property GLEAK, which is an admittance with default value 1E-12Mho. Setting this value to zero makes capacitors non leaky, as with traditional SPICE simulators.

Apart from the simulation of the innards of DRAM memory circuitry we cannot foresee any problems with this scheme, and it will save relative beginners from many strange error messages. In any case, real capacitors generally have leakage considerably more than a million megohms.

The ground net in a circuit can be defined either explicitly or implicitly. An explicit ground is defined by placing an unlabelled *Ground* terminal, as on the bottom end of RF2, above. You can also label a wire with the text **GND** if space is tight.

An implicit ground can be created through the use of a power rail, single pinned generator, loaded probe or a digital output, as well as by using any model that contains an internal grounded node.

Power Rails

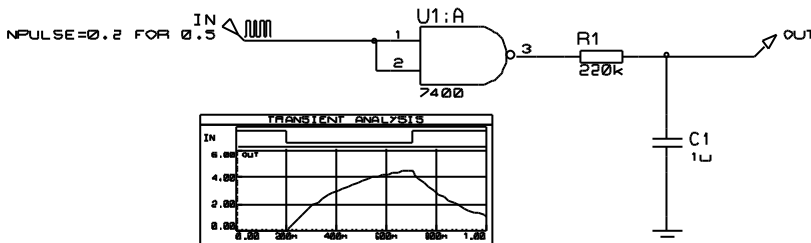
PROSPICE recognises a number of entities as power rails; the rules are as follows:

- Any net named GND or VSS is considered to be the ground reference. So far as PROSPICE is concerned, GND and VSS are aliases for the same thing.
- The net names VCC and VDD are considered to be the logic power rail, and will be treated as a logic '1' by DSIM. As with GND and VSS, PROTEUS VSM assumes these two names refer to the same net - if you really want split logic power supplies you must choose different net names.
- Power terminals with names of the form +5, -5 or +10V, -10V are taken as defining fixed power rails with reference to ground. The '+' and '-' symbols are crucial; a label such as 5.0V will not do.

Connecting two such labels with different values to the same net is an error.

The creation of a power rail also implicitly creates a ground.

An additional feature, added as part of the scheme for mixed mode simulation is that Logic IC models can be thought of as self powering. This enables you to draw circuits such as the one below and get sensible results, without having to draw in power supplies explicitly.



This works because the *Interface Model* defined for the 7400 includes a **VOLTAGE** property. This causes the creation of a 5V battery between the 7400's hidden power pins (VCC and GND) such that the VCC net acquires a potential of 5V. Other bits of circuitry connected to VCC, including the DAC object at the 7400 output then see this voltage.



For a further discussion of Interface Models, see page 119.

INITIAL CONDITIONS

Overview

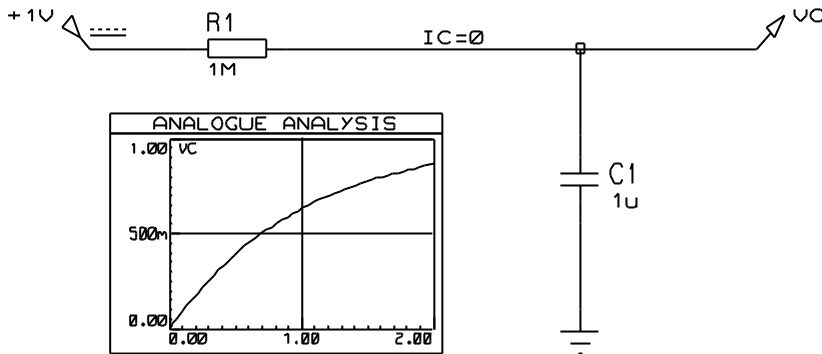
Whenever any type of simulation is performed, the first thing that PROSPICE does is to compute the operating point of the circuit - that is the steady state conditions that apply prior to any input signals being applied. There are two distinct modes of operation associated with computing these values:

- The operating point is found with all capacitors charged and inductors fluxed. In this case, a *Transient Analysis* will show the behaviour of the circuit as though power were applied sufficiently prior to time zero that by the start of the simulation everything had settled down. This is the default mode of operation, and given the ability to specify initial conditions for particular components and/or node voltages, serves for most purposes.
- The operating point is found with all capacitors short circuited and inductors open circuit. In this case, a *Transient Analysis* will show the behaviour of the circuit as though power were applied at time 0. This mode of operation can be selected by unchecking the *Compute Operating Point* option on the graph.

In either case, it is possible to specify the initial conditions for particular components or node voltages. This is especially useful for circuits which are oscillators, or for which circuit operation depends on particular capacitor being discharged at time zero. Indeed, the concept of a steady state operating point is meaningless for an oscillator, and the simulation may fail completely if some initial conditions are not given.

Specifying Initial Conditions for a Net

The easiest way to specify initial conditions is often to indicate the starting voltage for a particular net. In the circuit overleaf, this is achieved by adding a wire label with the text $IC=0$ to the probed net. Without this assignment, PROSPICE would compute the steady state value of the voltage on C1 i.e. 1 Volt and the graph would show VC as a horizontal straight line.



For nets which interconnect only digital components, you should use logic states for initial conditions - e.g. **1,0,H,L,HIGH,LOW,SHI,WHI,SLO,WLO** or **FLT** and assign them to the **BS** (Boot State) property. For mixed nets, you should specify an initial voltage - this will be automatically propagated as a logic level to the digital components.

Specifying Initial Conditions for Components

This option is only available when PROSPICE does not compute the initial operating point - i.e. when the *Initial DC Solution* option on the graph is not checked. In this case, all node voltages will be zero at time zero, except for nets with initial condition properties as described above. Under these conditions, it is then possible to specify initial conditions for particular components. For example, you could add the property

IC=1

to C1 in the previous example such that C1 started from its steady state value of 1 Volt.

Details of the IC properties supported by the various SPICE primitives are given in the chapter on modelling.

In some ways it is a shame that this option is not available when the operating point *is* computed, but this is how SPICE3F5 has been coded by Berkeley. However, we have added the **PRECHARGE** property as a remedy to this problem.

NS (NODESET) Properties

Occasionally, when a simulation fails to find the operating point, it can be useful to give SPICE a hint as to the initial values to use for particular nets. This is different from setting the initial conditions in that the value given is only used for the first iteration, and the net then

‘floats’ to whatever value the matrix solution converges at. It is, then, an aid to convergence and should not affect the actual operating point solution.

Such convergence hints can be specified using the **NS** net property, so placing a wire label with the text **NS=10** will suggest a starting value of 10 Volts for that net.

PRECHARGE Properties

A further option for specifying the initial conditions is the **PRECHARGE** property. This may be assigned to any capacitor or inductor in the circuit and specifies either voltage across the device or the current flowing through it respectively.

The **PRECHARGE** property is a Labcenter specific addition to SPICE, and differs from the **IC** property in that it is applied irrespective of whether the *Initial DC Solution* option is selected or otherwise.

TEMPERATURE MODELLING

The SPICE3F5 kernel provides extensive support for modelling the effects of temperature. The scheme operates as follows:

- There is a global property **TEMP**, which will, if nothing else is done, apply to all components in the circuit.
- The temperature of individual components can be specified by giving them their own **TEMP** property.
- Where parameters for a device model vary according to the temperature at which they were measured, this temperature can be specified globally and/or individually by **TNOM** properties.

Temperature modelling is provided for resistors, diodes, JFETs, MESFETs, BJTs, and level 1, 2, and 3 MOSFETs. BSIM models are expected to have been created for specific temperatures. No temperature dependency is implemented for digital primitives.

In addition, it is very important to realize that most equivalent circuit models for ICs will not model temperature effects correctly. This is because such models usually use ideal controlled sources and other macro modelling primitives, rather than containing an exact replica of the device internals. Once such primitives are used, it is unlikely that the model will show correct temperature dependent behaviour unless someone has taken the trouble to make it do so.

THE DIGITAL SIMULATION PARADIGM

Nine State Model

You might think that a digital simulator would model just high and low states, but in fact DSIM models a total of nine distinct states:

State Type	Keyword	Description
Power High	PHI	Logic 1 power rail.
Strong High	SHI	Logic 1 active output.
Weak High	WHI	Logic 1 passive output.
Floating	FLT	Floating output - high-impedance.
Undefined	WUD	Mid voltage from analogue source.
Contention	CON	Mid voltage from digital conflict.
Weak Low	WLO	Logic 0 passive output.
Strong Low	SLO	Logic 0 active output.
Power Low	PLO	Logic 0 power rail.

Essentially, a given state contains information about its polarity - high, low or mid-way -and its strength. Strength is a measure of the amount of current the output can source or sink and becomes relevant if two or more outputs are connected to the same net.

For example, if an open-collector output is wired through a resistor to VCC, then when the output is pulling low, both a *Weak High* and a *Strong Low* are applied to the net. The *Strong Low* wins, and the net goes low. On the other hand, if two tristate outputs both go active onto a net, and drive in opposite directions, neither output wins and the result is a *Contention* state.

This scheme permits DSIM to simulate circuits with open-collector or open-emitter outputs and pull up resistors, and also circuits in which tristate outputs oppose each other through resistors - a kind of poor man's multiplexer if you like. However, it is important to remember that DSIM is a digital simulator only and cannot model behaviour which becomes decidedly analogue. For example, connecting overly large resistors up to TTL inputs would work OK in DSIM but would fail in practice due to insufficient current being drawn from the inputs.

The Undefined State

Where an input to a digital model is undefined, this is propagated through the model according to what might be described as common sense rules. For example, if an AND gate has an input low, then the output will be low, but if all but one input is high, and that input is undefined then the output will be undefined.

- Edge triggered devices require a transition from a positively defined logic 0 to logic 1 (or vice versa) for an edge to be detected. Transitions from 0 or 1 to undefined and back are not counted as edges.
- The more complex sequential logic devices (counters, latches etc.) will often undefined inputs to a logic 0 or logic 1, according to the coding of their internal logic. This is not unlike real life!

Floating Input Behaviour

It is common, if not altogether sound practice to rely on the fact that unconnected TTL inputs behave as though connected to a logic 1. This situation can arise both as result of omitted wiring, and also if an input is connected to an inactive tristate output. DSIM has to do *something* in these situations since the internal models assume true logical behaviour with inputs expected to be either high or low.

This is catered for through the **FLOAT** property. This may be assigned either directly to a component, or an an *Interface Model*. In particular, the interface model for TTL parts has the assignment:

FLOAT=HIGH

which causes floating inputs to be interpreted as logic 1 levels.

To specify that floating inputs be taken as logic 0, use

FLOAT=LOW

Otherwise, floating inputs will be taken to be at the *Undefined* state. See page 115 for more information.

Glitch Handling

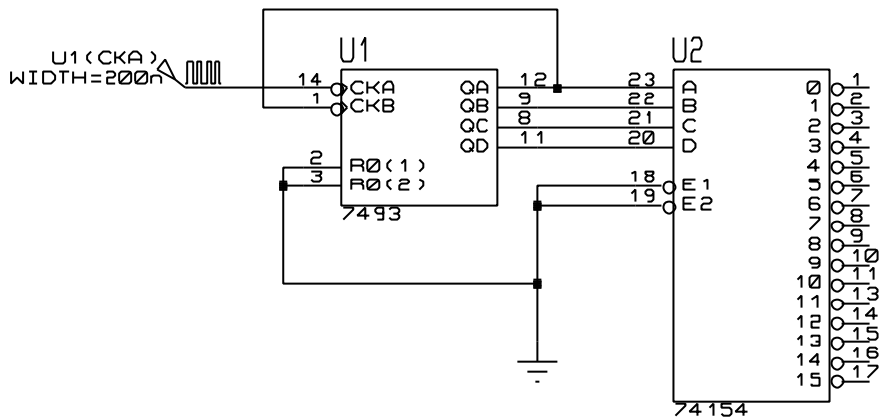
In designing DSIM we debated at great length how it should handle the simulation of models subjected to very short pulses. The fundamental problem is that, under these conditions, a major assumption of the DSIM paradigm - that the models behave purely digitally - starts to break down. For example, a real 7400 subjected to a 5ns input pulse will generate some sort of pulse on its output, but not one that meets the logic level specifications for TTL. Whether such an output pulse would clock a following counter is then a matter dependent on very much analogue phenomena.

The best one can do is to consider the extremes, namely:

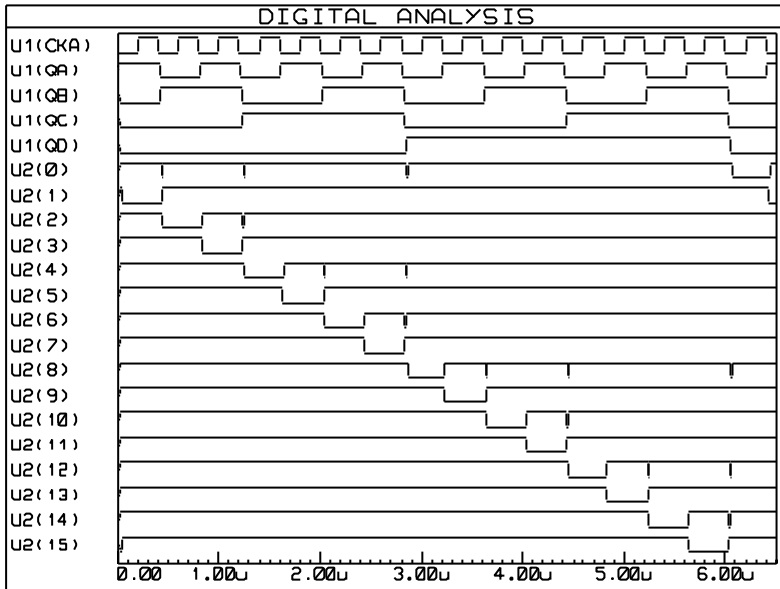
- A 1ns input pulse will not propagate at all.
- A 20ns pulse will come through nicely.

Somewhere in between, the gate will cease to propagate pulses properly and could be said to *suppress* glitches. This gives us the concept of a *Glitch Threshold Time*, which can be an additional property of the model along with the usual **TDLH** and **TDHL**.

Another subtlety concerns whether the glitch is suppressed at the input or the output of a model. To resolve this, consider a 4-16 decoder driven from a ripple counter as shown overleaf.



The outputs of the ripple counter are staggered, and thus the possibility arises of the decoder generating spurious pulses as the inputs pass through intermediate states. This situation is shown in the following graph:



The above graph was produced with TGQ=0 for the 74154

Taking the first glitch an example of the phenomenon, as U1(QA) falls for the first time, it beats the rise of U1(QB) and an intermediate input state of 0 is passed to the decoder for approximately 10ns. The question is whether the decoder can actually respond to this or not, and even more to the point, what would happen if the input stagger was only 1ns or 1ps? Clearly, in the last two cases the real device would not respond, and this tells us that we must handle glitches on the outputs rather than the inputs. This is because, in the above example, the input *pulses* are all relatively long and would *not* be considered glitches by any sensible criteria. Certain rival products make a terrible mess of this, and will predict a response even in the 1ps case!

The really interesting part of this tale is that, if you build the above circuit, it will probably not glitch. It is very bad design certainly, but the **TDLH** and **TDHL** of the '154 are around 22ns and this makes it a tall order for it to respond to a 10ns input condition. With the individual components we tried, no output pulses, other than perhaps the slightest twitches off the supply rails, were measurable.

To provide control over glitch handling, all the DSIM primitives offer user definable *Glitch Threshold Time* properties named **TGxx**, where **xx** is the name of the relevant output. Our TTL models are defined such that these properties can be overridden on the TTL components, and the values are then defaulted such that the *Glitch Threshold Times* are the average of the main low-high and high-low propagation delays. Setting the *Glitch Threshold*

Times to zero will allow all glitches through, should you prefer this behaviour. The graph, above, was thus created by attaching the property assignment **TGQ=0** to the 74154.

Finally, it is important to point out that if the *Glitch Threshold Time* is greater than either of the low-high or high-low propagation delays, then the *Glitch Threshold Time* will be ignored. This is because, after an input edge, and once the relevant time delay has elapsed, the gate output **must** change its output - it cannot look into the future and see whether another input event (that might cancel the output) is coming. Consider a symmetric gate with a propagation delay of 10ns and a *Glitch Threshold Time* of 20ns. At $t=0$ ns the input goes high and $t=15$ ns the input goes low. You might expect this to propagate, with the output going high at $t=10$ ns and low again at $t=25$ ns, so producing a pulse of width 15ns which would be suppressed, since it is less than the *Glitch Threshold Time*. The reason the pulse is not suppressed is that, at $t=10$ ns, the output **must** go high - it cannot remain low for a further 20ns on the off chance (as in our example) a second edge comes along so producing an output pulse it would need to suppress! Once the output has gone high at $t=10$ ns then the second edge (at $t=25$ ns) is free to reset it. You will need to think carefully about this to understand it.

MIXED MODE INTERFACE MODELS (ITFMOD)

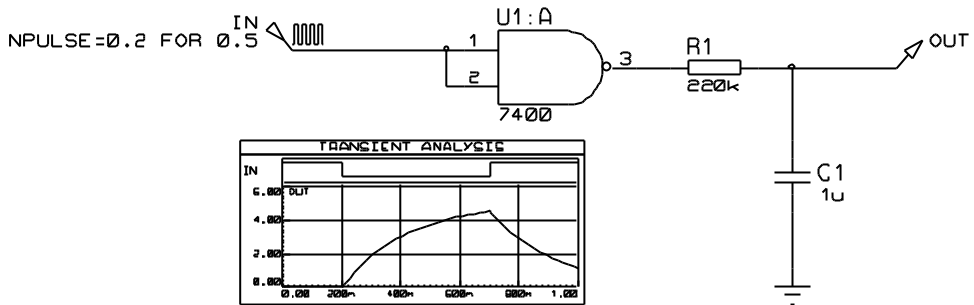
Overview

In designing our scheme for mixed mode simulation within PROSPICE, we gave considerable thought to the problem of how to specify the analogue characteristics of a digital device family. These characteristics include:

- The input and output impedances of the devices.
- The logic thresholds of device inputs.
- The voltage levels for high and low outputs.
- The rise and fall times of device outputs.
- The default logic state for floating inputs.

A scheme which involved specifying all these parameters for every device in the TTL libraries, say, would be extremely unwieldy.

In addition, a significant problem arises (for beginners, at least) in the specification of power supplies - there is a tendency to plonk down a circuit such as the one below and expect sensible results. The problem here, of course, is an implicit assumption that the 7400 has a 5V power rail obtained from its hidden power pins which connect to VCC/GND.



All these problems are solved by the introduction of the **ITFMOD** component property. This is very similar to the **MODEL** property in that it provides a reference to a set of property values but it also activates a special mechanism within the netlist compiler. Essentially this works as follows:

- For any device that has an **ITFMOD** property an additional model definition is called up during netlisting that will specify control parameters for ADC and DAC objects, and also the pin names of the positive and negative power supplies. In the above circuit, U1:A will have **ITFMOD=TTL**.
- Having obtained the names of the power supply pins (VCC, GND in this case), ISIS creates a special primitive and connects it across the power supply pins. ISIS names this object similarly to an object on the child sheet or model, so that in the above circuit, the power supply object will be called U1:A_#P.
- When PROSPICE simulates a mixed mode circuit, it creates ADC and DAC objects and considers them to 'belong' to the objects to which they connect. In the case of the circuit above, a DAC object will be created with the name U1:A_DAC#0000 because it forms the interface from U1:A's output.

The clever part is that on doing this, it also looks for a power supply interface object with the same name stem i.e. U1:A, and finds U1:A_#P. It then instructs U1:A_DAC#0000 to take its properties from U1:A_#P which in turn has inherited its properties from the model specified in the original **ITFMOD** assignment. Thus the DAC object operates with parameters defined for the **TTL** logic family.

- Each power interface object also contains a battery which will be assigned to the **VOLTAGE** property given in the interface model definition. The **TTL** interface model definition specifies **VOLTAGE=5V**.

This means that in the above circuit, a 5V battery gets inserted between VCC and GND, because these are the nets indicated by the power pins of the 7400 device.

- The batteries have a small internal impedance (1miliohm). This means that if you assign a real power rail to VCC/VDD (by placing a power terminal or voltage source) then this will override the level defined by the internal batteries - in the world of simulation, a large current flow through the batteries does not matter!

Using ITFMOD Properties

Existing interface models have been defined as follows:

TTL	Standard TTL (74 series)
TTLs	Low power Schottky TTL (74LS series)
TTLs	Standard Schottky TTL(74S series)
TTLHC	High Speed CMOS TTL (74HC series)
TTLHCT	High Speed CMOS TTL with TTL outputs (74HCT series)
CMOS	4000 series CMOS.
NMOS	Microprocessor type MOS circuits.
PLD	PLD type MOS circuits.

It follows that any new digital model can be assigned a device family by adding a property such as

ITFMOD=TTL

The family definitions are held in the file ITFMOD.MDF which is kept in the models directory.

Each definition can contain any or all of the properties defined for the ADC and DAC interface primitives. In addition, the following may be given:

V+	-	Name of the positive power supply pin.
V-	-	Name of negative power supply pin.
VOLTAGE	5V	Specifies the default operating voltage.
RINT	1mΩ	Specifies the impedance of the internal battery. A value of zero will disable the battery.
FLOAT	-	Specifies HIGH or LOW value for floating inputs.

Finally, it is worth pointing out that any specific property e.g. TRISE, can be overridden on the parent device, so if you want simulate a 4000 series IC with a slow rise time, you could add TRISE=10u directly to its property list.

PERSISTENT MODEL DATA

Certain simulator models, particular those for EPROMS, and micro-processors containing EEPROM or flash memory are able to 'remember' data between simulation runs, just like their real life counterparts. This action is facilitated by the **MODDATA** property. Persistent model

data blocks are stored in the DSN file, and so do not persist between PROTEUS sessions unless you save the design.

To reset the persistent model data to its initial condition, use the *Reset Persistent Model Data* command on the *Debug* menu.

TAPES AND PARTITIONING

Overview

A unique feature of the VSM system is its ability to divide a large design into one or more sections or *partitions* and simulate each individually.

There are two principal advantages associated with this:

- In a fully integrated CAD exercise, the circuit for the whole design of the product will contain some sections which you may not wish to or cannot simulate. To prevent the need for carving up the design, some mechanism is required for determining which components in the design are actually relevant to a given simulation experiment.

ISIS does this by considering the points being measured and, further back down the design, the points being driven by test sources and/or power rails.

- If the design consists of several stages, it will be a common requirement to see how later stages perform when driven by the outputs of earlier stages. Whilst this could be achieved by simulating all the stages together, this would tend to be unduly slow. Partitioning allows the results from simulating previous stages to be captured in tapes and played back as the input for subsequent stages.

ISIS contains logic to do this either under manual control or automatically. In the later case, it detects when the circuitry within given partitions has changed and re-simulates only those which have changed, or which are affected by those which have changed.

Single Partition Operation

Many simulation experiments will consist of testing a single section of a design in isolation from the rest. This will generally involve injecting a signal or signals at the input(s) of the section, and then monitoring its operation at points throughout it.

It should be self evident that signals can be injected by placing generator gadgets on appropriate input wires, and that circuit operation can be monitored by placing probes. However, these actions by themselves do not isolate the section under test from the rest of the design.

To do this, you need to set the *Isolate Before* checkboxes on the generators and the *Isolate After* checkboxes on the probes. Once this is done, only the section between the isolation points will be netlisted and simulated.

To work out which section to simulate, ISIS looks at the probes that have been assigned to the graph, and propagates through components and wires (including inter-terminal connections) outwards until one of the following conditions is met:

- There are no more components to process.
- An isolating probe or generator is reached.
- A tape input or output is reached.
- A power rail is reached. A net is made into a power rail by connected a POWER or GROUND terminal to it. A GND or VCC wire label will NOT do here.

Tape Objects

Tape objects have two distinct uses with the system:

- To define points at which it is reasonable to ignore circuitry to the right of when simulating circuitry to left. Such points are generally where low impedance outputs drive high impedance inputs. This can also be achieved by placing isolating probes.
- To provide the means to capture the output of one stage of a design, and use it to drive the next stage but without the overhead of simulating the first stage.

To Place a Tape:

1. Select the *Gadget Mode* icon and then the *Tape* icon.
2. Use the *Rotation* and *Mirror* icons to orient the tape according to how you want to place it. The little arrow symbol within the tape body designates the direction of the tape; in its normal orientation, the input is on the left and the output is on the right.
3. Place the mouse in the *Editing Window*, press down the left mouse button and drag the tape to the correct position. Release the mouse button.

You can place a tape directly on to an existing wire by placing it such that its connection point touches the wire. Alternatively you can place several tapes in a free area of your design, and wire to them later.

It is vital to place tapes only in sensible locations - that is where low impedance drives high. If you place tapes in other locations, you will change the operation of your circuit and invalidate any results that may then be produced.

Tape Modes

To give the maximum degree of user control, tapes have three modes:- AUTO, PLAY and RECORD. In the following discussions, we use the terms left and right in terms of the logical dependency of the partition tree, reading the design input⇒output, left⇒right.

AUTO mode

This is the default scheme and defines a mode of operation in which ISIS decides which partitions need re-simulating, and which can use previously stored data. For most situations that require the use of tapes, AUTO mode will prove to be the most useful.

This automatic detection is based on considering all the text that occurs within the sub-netlist related to the partition. It follows that if any part names, values, properties, wiring etc. in that partition is changed, a re-simulation will be performed unless a partition file already exists for that set of information.

Note also that ALL models, scripts, design global properties and so forth are included in the sub-netlist, so changing any such object will cause a re-simulation of all automatic partitions. ISIS does not delve into the question of whether particular models, scripts etc. are used by components in a particular partition. If you need to overcome this, you can use the manual RECORD and PLAY modes.

Note that a TAPE object in auto mode will be removed from the circuit if an interactive simulation is performed.

PLAY mode

This mode enables you to play a named file that you have previously recorded, either with a tape or by adding a **RECORD** property to a probe. The filename of the data to be played should be entered into the *Filename* field of the tape; you cannot use PLAY mode unless a filename has been entered.

When a tape is in PLAY mode, the circuitry to its left is disconnected and ignored, unless it includes probes which are also included on the current graph.

Do bear in mind that you can only play data that has been recorded for the type of analysis currently being performed. Attempts to do otherwise will result in an error.

RECORD mode

This mode causes the data present at the input of the tape to be recorded into the file named into its *Filename* field; you cannot use RECORD mode unless a filename has been entered.

Another effect of record mode is to force the partition to the tape's left to be re-simulated, irrespective of whether it has changed or not. If there is a partition to the right of the tape that

is probed, then this will also be re-simulated since it will be deemed to depend on the one to its left.

The RECORD mode for tapes is probably most useful in situations in which you want to record a waveform once, and then lock it as the input for further experiments on the right hand side of the tape.

SIMULATOR CONTROL PROPERTIES

Overview

There are a large number of parameters that affect the details of how a simulation is performed. These include things such as the maximum number of iterations allowed to find the operating point, the tolerances that determine when a solution has converged, the integration method used and so forth.

These options are common to all types of analysis and can be adjusted separately for each graph by editing it and clicking the *SPICE Options* button.

Tolerance Properties

This group of parameters determine how accurately SPICE will compute the solution. Higher accuracy is generally achieved at the expense of longer simulation times, and in some circumstance the circuit may fail to converge at all if you ask for too high a set of tolerances.

The most useful value here is the *Truncation Error Estimation* factor, or **TRTOL** in traditional SPICE nomenclature. If you have results which appear overly spiky, or suffer from overshoots you should try a lower value here.

The minimum conductance value, **GMIN**, defines the leakage of reverse biased semiconductor junctions and other theoretical points of infinite impedance. Reducing this value may help achieve convergence for circuits which fail to simulate although this may be at the expense of simulation accuracy. See page 135 for more information about convergence issues.

The leakage conductance, **GLEAK** is something we have introduced to enable solution of circuits with DC blocking capacitors. It defines the DC leakage of capacitors, and can generally be left alone unless you are simulating something unusual such as CMOS memory cells or the like.

Mosfet Properties

SPICE simulation of MOSFETs is based on the assumption that you are doing IC design, and consequently implements a scheme for scaleable geometries. In practice this means that there are a number of parameters which determine the default physical sizes for elements of the MOSFET devices. These are values are defined here.

In addition, some models have been created which rely on the behaviour of older versions of SPICE, and this MOSFET behaviour can be switched on or off here if you are using older MOSFET models.

Iteration Properties

The properties on the *Iteration* tab determine how SPICE deals with circuits that are hard to converge.

The integration method can be either *Gear* or *Trapezoidal*. The latter is provided mainly for backward compatibility with previous versions of SPICE since *Gear* integration generally gives more accurate results for a given number of time-steps. In *Gear* integration, high orders than 2 are possible; this involves SPICE using more of the past history of a time-point in predicting what happens at the next time-point.

When SPICE fails to get a convergent solution at the operating point, it tries two approaches: *Gmin Stepping* and *Source Stepping*. The number of steps tried in each method can be set here.

Three further options determine that maximum number of iterations that will be tried at each of the operating point, a step in a *Transfer* analysis, and a time-point in a *Transient Analysis*. Increasing these determines may help in getting a result out of a hard or near unstable circuit.

Finally, two options are given that may result in faster simulations. LTRA compaction applies only for circuits incorporating lossy transmission lines (LOSSYLINE model). The idea is that near identical values in the data pipeline get discarded so that this data points are processed. Bypassing unchanging elements is a general optimization that saves SPICE from re-computing the values of semiconductor devices whose node voltages have not changed since the last evaluation.

Temperature Properties

There are two global temperature properties: **TEMP** - the default operating temperature , and **TNOM** the parameter measurement temperature. **TEMP** defines the actual temperature of the circuit, whilst **TNOM** is the value at which temperature dependent device parameters are taken to have been measurement. For further discussion of temperature modelling in PROSPICE see page 114.

Digital Simulator Properties

TDSCALE, TDSEED, TDLOWER and TDUPPER

The **TDSCALE** variable is used to control the scaling of all timing properties used by models in the simulation run that have not been explicitly referenced by the model as non-scaleable. The **TDSCALE** variable can be assigned either a constant floating-point value or the keyword **RANDOM**.

If a constant value is specified, then all timing properties specified in models used in the simulation are multiplied by the value; a value less than 1.0 reduces the timing properties whilst a value greater than 1.0 increases them. For example, the variable assignment:

```
TDSCALE = 1.1
```

has the affect of extending all timing properties used in the simulation by 10%. The default value for **TDSCALE** is the constant value 1.0; this causes all timing properties to be used unmodified.

If the **TDSCALE** variable is assigned the keyword **RANDOM**, the DSIM engine will randomly scale each timing property by multiplying it by a random floating-point time-scaling value. The range of time-scaling values chosen by the simulation engine can be limited by assigning the **TDLOWER** and **TDUPPER** variables; these define the lowest and highest random floating-point scale values allowed respectively. The default values for **TDLOWER** and **TDUPPER** are 0.9 and 1.1 respectively which has the affect of limiting random scaling to within $\pm 10\%$.

The sequence of random values generated is said to be *pseudo-random* - that is, each successive value generated, whilst appearing to be random, is in fact determined by the previous value (and a complex formula). It follows that any sequence of random values is determined by a *seed* value and that for a given *seed* value the sequence of random values generated is always the same. The **TDSEED** property allows you to specify a seed value for the generation of random time-scaling values and thus guarantee that the sequence chosen from one simulation run to another is always the same. This avoids the problem of a timing design error based on random propagation delays appearing and then disappearing on successive simulation runs.

The **TDSEED** variable should only be assigned positive integer values in the range 1-32767. The default value for **TDSEED** is itself a random value (based on the date and time) and this provides a means for randomising the sequence of values generated from one simulation run to the next.

For example, the assignments:

```
TDSCALE = RANDOM
TDLOWER = 2.00
```

```
TDUPPER = 3.00  
TDSEED  = 723
```

have the affect of randomly increasing all timing properties by approximately 200-300% with a random sequence of scaling values. The seed value of 723 causes the same sequence of pseudo-random values to be generated for each simulation run.

INITSEED

The **INITSEED** property is used to seed the random initialisation value generator, used by DSIM primitive models whose initialisation properties have been assigned the **RANDOM** keyword.

As with the **TDSCALE** and **TDSEED** properties described above, whilst the sequence of values generated by the random initialisation value generator are random, the sequence as a whole is finite and determinate. The **INITSEED** property provides a means of selecting which sequence of random values is used by the DSIM simulator.

The **INITSEED** property should only be assigned positive integer value in the range 1-32767. The default value for **INITSEED** is itself a random value (based on the date and time) and this provides a means for randomising the sequence of values generated from one simulation run to the next.

TYPES OF SIMULATOR MODEL

How to tell if a Component Has a Model

PROTEUS is supplied with over 8000 library parts of which about 6000 have simulator models. The devices which do not have models are perfectly relevant for use in PCB design, and the notion that every part should have a model is quite untenable. It would imply, amongst other things, that we should create a model for the 68020 processor which is not exactly a trivial job.

So, for the purposes of circuit simulation, you need to be able to tell if a component you are using has a simulator model. In the first instance, an attempt at simulation will fail if it doesn't, generally with a message something like:

```
ERROR [PSM] : No model specified for 'U1'.
```

The error is detected in the partitioning (PSM) phase, because up until this point, it may be that the un-modelled component is not actually in the part of the circuit being simulated.

Occasionally you will get:

```
ERROR [U1] : Value '74F00' of VALUE not found in
             parameter mapping table.
```

This means that there is a model file for the device, but that you have changed the value to a part type that is not modelled by the MDF file. In the above example, we have changed the value of a 74LS00 gate (which is modelled) to a 74F00 gate, which isn't.

The type of simulator model (if any) available for a component is displayed at the top left of the preview window in the device library browser. For example, if you open the library browser and select the 74LS00 device in the 74LS library, you will see

```
Schematic Model [74NAND.MDF]
```

Further information about the various types of models is given in the following sections.

Primitive Models

A significant number of basic component types are built directly into PROSPICE. These device types are called *Primitives* and include resistors, capacitors, diodes, transistors, gates, counters, latches, memories and many more.

Primitive models require no extra files to be simulated, and are identified by the presence of a single PRIMITIVE property. Additional properties are specified directly within the component and are passed to PROSPICE via the netlist. For example, a resistor will have:

```
PRIMITIVE=ANALOG,RESISTOR
```

This identifies the part as a SPICE Resistor primitive.

The standard set of simulator primitives may be found in the ASIMMDLS and DSIMMDLS libraries. These parts provide context sensitive help for their properties, and examples of their use may be found in the VSM SDK Documentation.

Schematic Models

Where a more complex device is to be simulated, a common approach is to draw a circuit that mimics its action using simulator primitives. This circuit can be the actual internal electronics of the device, but more commonly will utilize ideal current sources, voltage sources and switches to speed things along.

A schematic model is specified with the **MODFILE** property, which by convention we make a read only property. For example, the 741 op-amp has the assignment:

```
MODFILE=OA_BIP
```

You will note from this, that the model file is able to model several devices – a feat which it achieves using a *Parameter Mapping Table*.

Schematic models are created by drawing a schematic in ISIS, and then compiling it with the *Model Compiler* to produce an MDF file. Further details of this process are provided in the VSM SDK documentation.

VSM Models

VSM models are really primitive models which are implemented in external DLLs as opposed to inside PROSPICE itself. They provide the means to simulate device functionality using the programming language of your choice, although it is generally easiest to use C++.

A VSM model will have both a PRIMITIVE property (because both ISIS and PROSPICE treat it as a primitive) and a MODDLL property which specifies the name of the DLL file in which the model's code resides.

For example, the 8052 model has

```
PRIMITIVE=DIGITAL,8052
MODDLL=MCS8051
```

Note that a model DLL can implement more than one primitive type – MCS8051.DLL implements a number of 8051 variants.

VSM models can also implement functionality that relates to animation, such that the electrical and graphical aspects of a component's operation can be combined in fairly astounding ways. The LCD display model is a good example of this.

Creating VSM models revolves around a number of C++ Interface classes (similar to COM). These are documented in the VSM SDK manual.

SPICE Models

Since PROSPICE is based on Berkeley SPICE3F5 it is directly compatible with standard SPICE models, and many of the components in the PROTEUS libraries are modelled using SPICE files obtained from component manufacturers. SPICE models can be specified either by SUBCKT blocks or by sets of parameters in a MODEL record. SUBCKT models will have the property assignments such as

```
PRIMITIVE=ANALOG,SUBCKT
SPICEMODEL=CA3140
```

whereas SPICE primitive model for a transistor might have the properties:

```
PRIMITIVE=ANALOG,NPN
SPICEMODEL=BC108
```

The model itself can be stored either in an ASCII file, or in a SPICE Model Library. The name of these files is specified by a either **SPICEFILE** or **SPICELIB** property.

Extensive detail on how to make use of manufacturers SPICE models is given in the chapter entitled USING SPICEMODELS on page 103.

TROUBLESHOOTING

THE SIMULATION LOG

Whenever a simulation is performed a *Simulation Log* file is created. The simulation log file contains all information, warning and error messages from both the simulator itself and from individual models. Where a message originates from a model, it is prefixed with the models component reference in square brackets, so you might see:

```
[U1] Loaded 26 files from PROGRAM.HEX
```

During an interactive simulation, the contents of this file can be displayed in a popup window from the *Debug* menu whilst for a graph based simulation, the file can be viewed by pointing at the graph and pressing CTRL+V.

In the case where the simulation fails completely, the log file will be displayed automatically in order than you can see the cause of the problem immediately.

NETLISTING ERRORS

Netlisting errors occur as a result of a problem when ISIS tries to create a netlist from the schematic - you will also encounter these if you try to export the schematic to ARES for PCB layout. Common ones are:

- Having two components with the same name, or unnamed components e.g. two resistors labelled R?.
- Badly formed script files such as **MAP ON** tables etc. Refer to the syntax definitions in the ISIS manual if you are can't spot the problem immediately.

LINKING ERRORS

Model linking is the process whereby ISIS calls up MDF files for components which are modelled by equivalent circuits. By far the most common problem is where the specified model file does not exist. The model file must be in the current directory, or in the *Module Path* as set on the *Set Paths* dialogue form.

Other common linker errors include:

- Value not found in parameter mapping table. This means that the part type - e.g. CA3140 is not listed in the mapping table of the specified model file. Model files such as OA_MOS.MDF are design to model several different components using the same circuit but with different values. This error means that the model file specified does not have

parameters for the device type you are using - you can edit the MDF file with a text editor to see which devices are modelled.

- Unresolved module pin. This is a warning rather than an error, and means that the parent component body has a pin which is not present in the model. This is often OK - for example, most op-amp models do not model the offset null pins, but it can be a mistake, and if a new model doesn't work this is a common cause - especially if there are also 'No DC path to Ground' warnings.

PARTITIONING ERRORS

Partitioning is the mechanism by which ISIS decides which part(s) of the circuit need to be simulated. Problems that can occur here are:

- Cyclic dependencies detected. This means that an arrangement of tapes is such that the partitions behind and in front of a tape are mutually dependent. Assuming that you have correctly set any *Isolate Before* and *Isolate After* flags on probes and generators your simplest action here will probably be to delete the tape object(s) and simulate the whole circuit in one go.
- No model specified - this means that the component indicated does not have a **PRIMITIVE**, **MODFILE** property, and has appeared in that part of the circuit which needs to be simulated. If the device is irrelevant (e.g. a connector) then you must specify **PRIMITIVE=NULL**, otherwise you need a model!

See page 128 for more information about simulator model types.

SIMULATION ERRORS

Simulation errors are generated by PROSPICE rather than ISIS, and therefore occur after a netlist file has been successfully generated. Common problems that get detected here include:

- Device type not recognized. This means you have specified a primitive type that is not supported, or that a model file has used one.
- No DC path to ground. This is discussed further under **GROUND AND POWER RAILS** on page 109.
- Could not find probe - you have tried to reference a probe or voltage generator that does not exist. Remember that you *must* use an **Iprobe** object from **ASIMMDLS.LIB** - you cannot reference a current probe gadget.
- Cannot open SPICE source file. The source file specified in a **SPICEMODEL** property cannot be located. It should be in the current directory or in the *Module Path* as set on the *Set Paths* dialogue form.

- Cannot find library model. The SUBCKT or MODEL you have specified does not exist in the specified library or on disk.
- Model DLL not found. The specified VSM model DLL cannot be located. It should be in the current directory or in the *Module Path* as set on the *Set Paths* dialogue form.

CONVERGENCE PROBLEMS

This final set of errors relate to what happens if SPICE itself fails to simulate the circuit. There are basically three error messages that indicate this:

- Singular matrix. This is akin to have more unknowns than equations and usually relates to circuits which are mis-drawn, or in which some initial conditions need to be given in order to define the starting state.

This error will often be preceded by “No DC path to ground” warnings, and you need to investigate the wiring around the pins listed after this message. If part of your circuit is not ground, the simulator can resolve its voltage relative to ground - it’s as simple as that.

- To many iterations without convergence. This means that circuit solution is unstable. Circuits with VSWITCH or CSWITCH primitives can create this condition easily, but any circuit whose transfer function is discontinuous can give SPICE serious problems.
- Timestep too small. This means that the circuit has switched in such a way that advancing the time even by very small amounts (typically 1E-18s) still does not produce an acceptably small change in circuit voltages.

Often, this is caused by a badly designed model, or by not supplying sufficient parameters to a diode or transistor model. In a particular, if the junction capacitance values are not chosen correctly, these devices will exhibit zero switching times which can lead directly to this error message.

Most convergence errors are due to badly drawn circuits or incorrect models - time after time we have had circuits sent in that ‘won’t simulate’ only to find that something isn’t connected. Please check the simulation log for clues, and re-check your circuit before jumping to the conclusion that PROSPICE is at fault. *Where 3rd party SPICE or VSM models have been used, we cannot spend time debugging them unless can supply a simple circuit demonstrating the problem.*

Beware also of using 3rd party SPICE models which use features not supported in standard SPICE 2 or SPICE 3. Models developed for PSPICE™ can include all manner of elements and syntax constructs that are not standard SPICE.

Oscillators cause special problems because the initial solution for the operating point will fail. After all, an oscillator has no steady state! Use **IC** , **NS** or **OFF** properties to define a starting state as discussed under *Initial Conditions* on page 112.

If the problem really is numerical convergence, you can try the following tactics:

- Increase the value of **GMIN**. This is a leakage resistance for reverse biased semiconductor junctions, and lower values make the circuit look more and more like a network of resistors (which will always solve). But this is at the expense of accuracy. The default is 1E-12; values above 1E-9 will give fairly meaningless results.

Note in any case, that SPICE3F5 will try what is called *GMIN stepping* if at first the circuit will not converge. This means that a large GMIN is used to find an initial solution, and the value is then ramped back to its original value in order to maintain accuracy.

- Increase the value of **ABSTOL** and/or **RELTOL**. These values control the accuracy that is required for the simulation to be deemed to have converged. However, the larger you make these tolerances, the less accurate the results will actually be.
- If the circuit uses op-amps, try specifying **MODFILE=OA_IDEAL** instead of a specific device type - this model is much easier to simulate.
- Lower the value of **TRTOL**. This will make SPICE use smaller timesteps so it will be less likely to ‘lose’ a convergent solution, but at the expense of longer run times. This will only work if the simulation has failed part way through a transient analysis.

You should also try reducing **TRTOL** if plotted traces look ‘spiky’, or contain mathematical noise. This often manifests itself as oscillation of a value after a rapid level transition.

A

Aborting a Simulation	15
AC Sweep Analysis	79
Active Components	24
Actuators	24
Add Trace Command	14, 61, 62
Add/Remove Source Files command	48
Ammeter	29
Analogue Analysis	68
Analogue Traces	70
Analyses	
AC Sweep	79
Analogue	68
Audio	85
Conformance	87
DC Sweep	77
DC Transfer Curve	80
Digital	71
Distortion	82
Fourier	84
Frequency	75
Interactive	86
Mixed Mode	74
Noise	81
Power	70
Transient	68
Animation Control Panel	23
Animation timestep control	26
Assemblers	49
Audio Analysis	85
Audio Generator	99

B

Bode Plot	75
Breakpoints	54, 55, 56
Build All command	48
Bus Traces	73

C

Capacitors	
initial conditions	114
leakage value	109

Code Generation Tools	49
Coloured Wires	25
Compilers	49
Components	
Linking To SPICE Models	103
PRIMITIVE Property	65
Conformance Analysis	87
Convergence	69, 113, 126, 135
CPU load	26
Current	
displaying wire arrows	25
Current Arrows	25
Current Breakpoint	56
Current Generators	95
Current Graph, The	15, 60
Current Probes	17, 101
Cursors	
on graphs	16
on Logic Analyser	34

D

Data Driven Generator	99
DC Generator	95
DC Operating Point	91
DC Sweep Analysis	19, 77
DC Transfer Curve Analysis	80
DDE	50
DDX program	49, 53
Debug Data Extractor	49, 53
Define Code Generation Tools command	49
Demand Drive Simulation	63
Digital Analysis	71
Digital Breakpoint	56
Digital Generators	100
Digital Simulation	
Floating Input Behaviour	116
Glitch Handling	116
Glitch Threshold Time Properties	118
Initial Conditions	71
State Contention Behaviour	115
Termination Conditions	72
Undefined State	115
Digital Traces	73

Distortion Analysis	82
DSIM	71
control properties	127

E

Earthing	27
Edit Generator Dialogue Form	94
Edit Graph Trace Dialogue Form	63
Editing	
Generators	12, 94
Graph Traces	63
Graphs	15, 60
Probes	13
EEPROMs	121
Embedded Workbench	50
Error Handling	66, 133, 134
Event Driven Simulation	71
Exponential Pulse Generator	97

F

Fast Fourier Transform	84
File Generator	99
Fourier Analysis	84
Frames per second	26
Frequency Analysis	18, 75
Frequency Reference	18, 76
Function Generator	34

G

Generators	93
Adding To Graphs	13, 58, 61
Analogue Pulse	96
Audio Signal	99
Automatic References	12
DC	95
Digital	100
Editing	12
Exponential Pulse	97
File Based	99
Frequency Reference	18, 76
Piecewise Linear	98
Placing	12, 57
SFFM	97
Sine	95
Using To Partition A Design	123
WAV File Based	99

GETSPICE.EXE	107
GLEAK property	109
Glitch Threshold Times	117, 118
GMIN Stepping	136
GND netname	71, 109, 111

Graph Based Simulation

Aborting	15
How It Works	65
Report Log	15, 21, 82
Simulation Run Time	15

Graphs

AC Sweep	79
Adding Generators	13, 58
Adding Probes	13, 58
Adding Traces	13, 60
Analogue	69
Audio	85
Colour of Traces	63
Conformance	88
Current Graph	15, 60
DC Sweep	78
DC Transfer Curve	80
Digital	72
Distortion	83
Editing	15
Fourier	84
Frequency Graphs	76
Interactive	86
Maximising	16
Minimising	16
Mixed Mode	75
Moving Traces	14, 16
Noise	82
Parts Of	16
Placing	13, 58
Removing Traces	14
Resizing and Moving Graphs	13
Simulating	64
Status Bar	17
Tagging Traces	63
Taking Measurements From	16
Using Cursors With	16
Ground	27, 109

H

Harmonic Distortion	82
---------------------	----

High Impedance Points 27

I

IAR Embedded Workbench 50

IC net property 112

IDE 50

Indicators 24

Inductors

 initial conditions 114

Initial Conditions 112

Initial DC Solution Checkbox 113

INITSEED control property 128

Input Impedance Plots 77

Interactive Analysis 86

Interactive Simulation 24

Interface Models 111, 119

Intermodulation Distortion 82

Isolate Before Checkbox 95

ITFMODE property 119

K

Keil uVision2 50

Knobs 45

L

Leaky Capacitors 109

Logic Analyser 32

Logic states 25

LST files 49, 53

M

Magnitude Axis 77

MAKE program 49

Manufacturers' Model Files 103

Mathematical Noise 136

Maximising Graphs 16

Measurements

 using Graphs 16

 using Logic Analyser 34

 using Oscilloscope 29

 using Voltmeters and Ammeters 29

Memory windows 52, 55

Meters 29

Microprocessors 47

Minimising Graphs 16

Mixed Mode Simulation 74, 119

MODDATA property 121

MODDLL property 130

MODEL card 106, 130

Model Files 65

 SPICE 103

Model types 128, 129, 130

Modelling

 Using SPICE Models 103

MODFILE property 129

Module Path Directory 65

MOSFET default dimensional properties 125

Moving

 Graphs 13

 Traces On Graphs 14, 16

MP-LAB 50

N

Netlists

 Compilation Of 65

 Error Handling 133

 Linking Of 65

No DC Path to Ground 109

No model specified 128

Nodal Analysis 68

NODESET option 113

Noise Analysis 20, 81

NS property 113

O

Object Code 48

Object Selector 13, 60, 94

Operating Point 69, 91, 112

 In Mixed Mode Simulation 74

Oscillators 112

Oscilloscope 29

Output Impedance Plots 77

P

Parameter mapping table 128

Partitions 65, 122, 134

Pattern Generator 36

Persistent Model Data 121

Phase Axis 77

Piecewise Linear Generator 98

Pin logic states 25

Pin name translation	105	SFFM Generator	97
Placing		Signal Generator	34
Generators	12, 57, 94	Simulate Command	15
Graphs	13, 58, 60	Simulation Log	15, 21, 64, 66, 82, 133
Probes	12, 57, 101	Simulator Control Properties	125
Tapes	123	Simulator models	128
Popup Windows	52	Sine Generator	95
Power dissipation	70	Single Frequency FM Generator	97
Power Supplies	111	Single Step Time	26
in Mixed Mode Simulation	119	Single stepping	53
Role In Partitioning	123	Singular Matrix	135
Precharge Property	114	SML files	107
Primary cursor	16	Source Code Control System	47
PRIMITIVE Component Property	65	Source code editor	50
Primitive models	129	Source code windows	52, 53
PRIMITIVE property	129	Source Files	48
Probes	101	Source Level Debugging	49, 53
Adding To Graphs	13, 58, 61	SPICE model libraries	107
Current Probes	17	SPICE models	103, 130
Editing	13	SPICEFILE property	131
Placing	12, 57	SPICELIB property	107, 131
Using To Partition A Design	123	SPICEMODEL property	105, 107, 130
Properties		SPICEPINS property	105
Glitch Threshold Time Properties	118	SRCEDIT	50
ITFMODE Component Property	120	Status windows	52
PRIMITIVE Component Property	65	Step commands	53
Pulse Generator	96	Stopping a Simulation	15
PUTSPICE.EXE	107	SUBCKT definition	104, 130
PWI File	53	Swept Variable Analysis	19

R

Reference Generators	
Frequency	18
On Frequency Graphs	76
Register Windows	52
Reset Persistent Model Data command	121
Resizing Graphs	13
Results Path Directory	66
Rotary Dials	45
RTDBREAK object	56
RTIBREAK object	56
RTVBREAK object	56

S

Schematic models	129
Setup External Text Editor command	50

T

Tapes	123
TDLOWER control property	127
TDSCALE control property	127
TDSEED control property	127
TDUPPER control property	127
TEMP property	114, 126
Temperature	114, 126
Text Editor	50
Time Domain Analysis	68, 71
Timestep per frame	26
Timestep too small	135
TNOM property	114, 126
Toggle breakpoint command	54
Total Harmonic Distortion	82
Trace Expressions	62, 70

Traces		Wire colour	25
Adding To Graphs	60		
Analogue Traces	16	Z	
Bus Traces	73		
Deleting	14, 16	Zero time delay in models	72
Digital Traces	73		
Editing	63		
Moving On Graphs	14, 16		
Tagging	14, 16, 63		
Untagging	14, 63		
Using Expressions With	58		
Transient Analysis	68, 71		
Translating pin names for SPICE	105		
TRTOL property	136		
Tutorials			
Graph Based Simulation	10		
Interactive Simulation	3		
Types of model	128		
U			
UIC option	112		
UltraEdit	50		
V			
Value not found	128		
Variables Window	54		
VCC netname	71, 111		
VCR controls	23		
VDD netname	71, 111		
View Log command	15		
Viewing Simulation Logs	15, 21		
Virtual Instruments	29		
Voltage			
displaying as wire colour	25		
Voltage Breakpoint	56		
Voltage Probes	101		
VOLTAGE Property	111		
Voltmeter	29		
VSM models	130		
VSS netname	71, 111		
W			
Watch window	55		
WAV Files	85, 99		
Windows	52		
Wire arrows	25		

