

gpsim

T. Scott Dattalo

05 JUNE 2005

Contents

1	gpsim - An Overview	5
1.1	Making the executable	5
1.1.1	Make Details - ./configure options	5
1.1.2	RPMs	6
1.1.3	Windows	6
1.2	Running	6
1.3	Requirements	7
2	Command Line Interface	8
2.1	attach	9
2.2	break	10
2.3	clear	11
2.4	disassemble	12
2.5	dump	12
2.6	echo	12
2.7	frequency	12
2.8	help	13
2.9	icd	13
2.10	list	13
2.11	load	13
2.12	macros	14
2.13	module	15
2.14	node	16
2.15	processor	17
2.16	quit	17
2.17	run	17
2.18	step	18

<i>CONTENTS</i>	2
2.19 symbol	18
2.20 stimulus	18
2.21 stopwatch	19
2.22 trace	19
2.23 version	19
2.24 x	20
3 Graphical User Interface	21
3.1 Main window	21
3.1.1 Menus	21
3.1.2 Buttons	21
3.1.3 Simulation mode	22
3.2 Source Browsers	22
3.2.1 .asm Browser	22
3.2.2 Opcode view - the .obj Browser	23
3.3 Register views	24
3.4 Symbol view	25
3.5 Watch view	26
3.6 Stack viewer	26
3.7 Breadboard	27
3.8 Trace viewer	27
3.9 Profile viewer	28
3.10 Stopwatch	29
4 Controlling the Flow: Break Points	30
4.1 Execution Break Points	30
4.1.1 Invalid Instruction Break Points	30
4.2 Register Break Points	31
4.3 Cycle Break Points	31
5 Trace: What has happen?	32
6 Simulating the Real World: Stimuli	34
6.1 How They Work	34
6.1.1 Contention among stimuli	35
6.2 I/O Pins	35
6.3 Asynchronous Stimuli	36
6.3.1 Analog Asynchronous Stimuli	37

<i>CONTENTS</i>	3
7 Modules	38
7.1 gpsim Modules	38
7.2 Writing new modules	39
8 Symbolic Debugging	40
9 Macros	41
10 Hex Files	42
11 The ICD	43
12 Theory of Operation	45
12.1 Background	45
12.2 Instructions	45
12.3 General File Registers	46
12.4 Special File Registers	46
12.5 Example of an instruction	46
12.6 Trace	48
12.7 Breakpoints	48

Introduction

gpsim is a full-featured software simulator for Microchip PIC microcontrollers distributed under the GNU General Public License (see the COPYING section).

gpsim has been designed to be as accurate as possible. Accuracy includes the entire PIC - from the core to the I/O pins and including ALL of the internal peripherals. Thus it's possible to create stimuli and tie them to the I/O pins and test the PIC the same way you would in the real world.

gpsim has been designed to be as fast as possible. Real time simulation speeds of 20Mhz pics are possible.

gpsim can be controlled from either a graphical user interface (GUI), a command line interface (CLI) or by a remote process. Typical debugging features like breakpoints, single stepping, disassembling, memory inspect & change, and so on are all supported. In addition, complex debugging features like real time tracing, assertions, conditional breaks, and plugin modules to name a few are also supported.

Chapter 1

gpsim - An Overview

If you don't care to wade through details, this chapter should help you get things up and running. The INSTALL and README files will provide more up-to-date information than this document, so please refer to those first.

1.1 Making the executable

gpsim's executable is create in a manner that's consistant with many of the other open source software:

command	description
<code>tar -xvzf gpsim-x.y.z.tar.gz</code>	expand the compressed tar file
<code>./configure</code>	Create a 'makefile' unique to your system
<code>make</code>	compile gpsim
<code>make install</code>	install gpsim

The last step will require root privileges.

1.1.1 Make Details - ./configure options

gui-less

The default configuration will provide a gui (graphical user interface). The cli (command line interface) is still available, however many people prefer just to use the cli. These hardy souls may build a command-line only interface by configuring gpsim:

```
./configure --disable-gui
```

debugging

If you want to debug gpsim then you'll probably use gdb. Consequently, you'll want to disable shared libraries:

```
./configure --disable-shared
```

This will create one, huge monolithic executable with symbolic information.

1.1.2 RPMs

gpsim is also distributed in RPM form. In recent versions, there are two RPMs: `gpsim-devel` and `gpsim`. Both of these must be installed. There is also a RPM for the source code. This can be used to build a binary RPM unique to your system. Please see the latest `INSTALL` and `README` for the most up to date information.

1.1.3 Windows

gpsim runs on Windows too. Borut Razem maintains the `gpsim` Windows web site:

<http://gpsim.sourceforge.net/gpsimWin32/gpsimWin32.html>

You can find detailed instructions there for installing `gpsim` and its dependencies. Snapshots can be found:

<http://gpsim.sourceforge.net/snap.php>

1.2 Running

The executable created above is called: `gpsim`. The following command line options may be specified when `gpsim` is invoked.

```
gpsim [-?] [-p <device> [<hex_file>]] [-c <stc_file>]
  -p, --processor=<processor name>    processor (e.g. -pp16c84 for the 'c84)
  -c, --command=STRING                startup command file
  -s                                  .cod symbol file
  -L, --                               colon separated list of directories to
                                       search.
  -v, --version                        gpsim version
  -i, --cli                            command line mode only
  -d, --icd=STRING                    use ICD (e.g. -d /dev/ttyS0).
Help options:
  -?, --help                           Show this help message
  --usage                               Display brief usage message
```

Typically `gpsim` will be invoked like:

```
[My-Computer]$ gpsim -s mypic-program.cod
```

(The *[My-Computer]*\$ text is an example of a typical bash command prompt - you'll only type the text after this prompt). This loads the .cod file generated by gputils.

Under Windows, gpsim can also be invoked by navigating through the Start/Program menu. This will open a DOS window to provide access to the command line interface. It's also possible to open a DOS window (or CygWin bash session) and invoke gpsim from there.

1.3 Requirements

gpsim has been developed under Linux. It should build and run just fine under the popular Linux distributions like Redhat. gpsim has also been ported to the MAC, MicroSoft Windows, Solaris, and BSD. Two packages gpsim requires that may not be available with all Linux distributions are readline and gtk (the gimp tool kit). The ./configure script should tell you if these packages are not installed on your system or if the revisions that are installed are too old.

There are no minimum hardware requirements to run gpsim. Faster is better though!

gputils, the gnupic utilities package, is also very useful. gpsim will accept straight hex files, but if you want to do any symbolic debugging then you'll want to use the .cod¹ files that gputils produces. The .cod files are in the same format as the .cod files MPASM² produces.

¹.cod files are symbol files that were created by ByteCraft and are used by Microchip.

²MPASM is Microchip's Assembler.

Chapter 2

Command Line Interface

The command line interface is fairly straight-forward. The table below summarizes the available commands. Brief descriptions of these commands can also be displayed by typing *help* at the command line.

command	summary
attach	Attach stimuli to nodes
break	Set a break point
bus	Add or display node busses
clear	Remove a break point
disassemble	Disassemble the current cpu
dump	Display either the RAM or EEPROM
frequency	Set processor frequency
help	Type help "command" for more help on a command
icd	In Circuit Debugger support.
list	Display source and list files
load	Load either a hex or command file
log	Log/record events to a file
node	Add or display stimulus nodes
module	Select & Display modules
processor	Add/list processors
quit	Quit gpsim
reset	Reset all or parts of the simulation
run	Execute the pic program
set	display and control gpsim behavior flags
step	Execute one or more instructions
stimulus	Create a stimulus
stopwatch	Measure time between events
symbol	Add/list symbols
trace	Dump the trace history
version	Display gpsim's version
x	examine and/or modify memory

The built in 'help' command provides additional online information.

2.1 attach

```
attach node1 stimulus1 [stimulus2 stimulus_N]
```

attach is used to define the connections between stimuli and nodes. At least one node and one stimulus must be specified. If more stimuli are specified then they will all be attached to the node examples:

```
gpsim> node n1 # Define a new node.
gpsim> attach n1 porta4 portb0 # Connect two I/O pins to the node.
gpsim> node # Display the new "net list".
```

2.2 break

The break command is used to set and examine break points. New break points are assigned a unique number. This number can be used to query or clear the break point. Break points halt the simulation when the condition associated with them is true. Break points are ignored during single stepping.

Examining break points

```
break [bp_number]
```

Break points can be examined by typing the break command without any options. Specific breaks can be queried by specifying the break point number.

Program Memory/Execution breaks

The most common break point is an execution break point. This one halts execution whenever the program counter reaches the address at which the break point is set. The syntax is:

```
break e|r|w ADDRESS [expr]
```

The simulation halts when the address is executed, read, or written. The ADDRESS can be a symbol or a number. If the optional expression is specified, then it must evaluate to true before the simulation will halt. The read and write options only apply to those processors that can manipulate their own program memory.

Register Memory breaks

gpsim can also associate break points with register accesses. This is useful for capturing bugs that stomp on RAM. E.g. you can say something like “halt execution whenever bit 4 of register 42 is cleared”. The command line syntax is:

```
break r|w REGISTER [expr]
```

The simulation halts when REGISTER is read or written and the optional expression evaluates to true. There are two styles of expressions supported. One involves only expressions of the REGISTER, the other is completely arbitrary. The examples below illustrate the differences.

Here’s an example of a register write break. This one will halt the simulation if any value is written to the variable named *temp1*.

```
break w temp1
```

Here the write is conditioned to happen for only a certain value:

```
break w temp1==0x22
```

Here the condition applies to specific bits:

```
break w temp1 & 0b11110000 == 0b11000000
```

This one breaks only if the hex digit 'C' is written to the upper nibble of temp1.

Boolean Expressions

Sometimes it's necessary to specify an auxiliary condition with a break point. For example, there may be a temporary variable that is shared throughout the code. You may wish to trap writes to that variable only while executing a specific subroutine. For example, the following break point triggers when temp1 is written and while the program counter is in between the labels *func_start* and *func_end*:

```
break w temp1 (pc >= func_start && pc < func_end)
```

TIP: Use this type of break point if you suspect an interrupt routine is over writing a variable.

Another situation is one where you wish to trap writes to a variable only if some other variable is a certain value:

```
break w temp1 (CurTask & 0x0f != 0b101)
```

If the firmware writes to the variable temp1 then the simulation will halt if the lower nibble of CurTask is not equal to 5.

Attribute Breakpoints

gpsim also supports a concept of *attribute breakpoints*. Attributes are parameters that gpsim and its modules expose to the user interface. For example, the simulator stopwatch exposes attributes which support breakpoints. This feature is intended mainly for module writers to provide a mechanism for allowing the user to control the module.

2.3 clear

```
clear bp_number
```

The clear command is used to clear break points. The break point number must be specified. The *break* command without any arguments displays all of the currently defined break points. This can be used to ascertain the break point number. Once cleared, a break point is deleted.¹

¹A break point disable/enable feature has been discussed and may be added a future date.

2.4 disassemble

```
disassemble [[begin:end] | [length]]
```

The disassemble command decodes the program memory opcodes into their standard mnemonics. With no options, the *disassemble* command disassembles instructions surrounding the current program counter:

```
gpsim> disassemble
current pc = 0x1c
    0012 2a03 incf reg3,f,0
    0014 0004 clrwdt
    0016 5000 movf reg,w,0
    0018 1001 iorwf reg1,w,0
    001a 1002 iorwf reg2,w,0
==> 001c 1003 iorwf reg3,w,0
    001e e1f4 bnz $-0x16 ;(0x8)
    0020 d7ff bra $-0x0 ;(0x00020)
```

With a single numeric option, the disassemble command will

2.5 dump

```
dump [r | e]
```

```
dump r or dump with no options will display all of the file
registers and special function registers.
dump e will display the contents of the eeprom (if the pic
being simulated contains any)
```

See the 'x' command for examining and modifying individual registers.

2.6 echo

The echo command is used like a print statement within configuration files. It just lets you display information about your configuration file.

2.7 frequency

This command sets the clock frequency. By default gpsim uses 4 MHz as clock. The clock frequency is used to compute time in seconds. Use this command to adjust this value. If no value is provided this command prints the current clock. Note that PICs have an instruction clock that's a forth of the external clock. This value is the external clock.

2.8 help

By itself, help will display all of the commands along with a brief description on how they work. 'help <command>' provides more extensive online help. The help command can also display information about attributes.

2.9 icd

icd [open <port>]

The open command is used to enable ICD mode and specify the serial port where the ICD is. (e.g. "icd open /dev/ttyS0"). Without options (and after the icd is enabled), it will print some information about the ICD.

2.10 list

```
list [[s | l] [*pc] [line_number1 [,line_number2]]]
Display the contents of source and list files.
Without any options, list will use the last specified options.
list s will display lines in the source (or .asm) file.
list l will display lines in the .lst file
list *pc will display either .asm or .lst lines around the pc
```

The list command allows you to view the source code while you are debugging.

2.11 load

The load command is used to load either hex, configuration, or .cod files. A hex file is usually used to program the physical part. Consequently, it provides no symbolic information. .cod files on the other hand, do provide symbolic information. The only reason to use a hex file is when there's no .cod file available.

The syntax for loading source code files is:

```
load [procestortype] file
```

gpsim will automatically determine if the file is a .hex or .cod file. The optional procestortype allows one to override the processor specified in a .cod file.

Configuration files are script files containing gpsim commands. These are extremely useful for creating a debugging environment that will be used repeatedly.

2.12 macros

Macros are defined like:

```

    name macro [arg1, arg2, ..., argN]
        macro body
    endm

```

And they're invoked by:

```

    name param1, param2, ..., paramN

```

Macros are a way of collecting several parameterized commands into one short command. The first line of a macro definition specifies the macro's name and optional arguments. The *name* is used to invoke the macro. The arguments are text string place holders. When a macro is invoked, the parameters are aligned with the arguments. I.e. *param1* in the invocation can be thought of being assigned to *arg1* in the definition. The parameters replace the arguments in the macro body.

In the following example, a variable or attribute called *mac_flags* is being manipulated in an expression. The arguments *add* and *mask* appear in the macro body and provide a parameterized way of manipulating this expression.

```

    mac_exp macro add, mask
        mac_flags = (mac_flags+add) & mask
    endm

```

Note that the indentation is arbitrary. The macro is invoked by:

```

    mac_exp 1, 0b00001111 # increment the lower nibble

```

The parameter *add* is replaced by the number *1* while *mask* is replaced with the binary number *0b00001111*. The invocation turns into the *gpsim* command:

```

    mac_flags = (mac_flags+1) & 0b00001111

```

Nested Macros

The macro body can contain any *gpsim* command. Of particular interest are macro invocations within other macros. Here's another macro that invokes the one defined above.

```

    # Nested macro example
    mac1 macro p1, p2
        run
        mac_exp p1, p2
    endm

```

And it could be used like:

```
mac1 1,      0b00001111  # test lower nibble
mac1 (1<4), 0b11110000  # test upper nibble
```

The first invocation starts the simulator by executing a *run* command. When a break point is encountered, control returns to the command line and the *mac_exp* macro is invoked.

Displaying Defined Macros

All currently defined macros can be displayed by typing the macro command without a name or arguments:

```
gpsim> macro
mac1 macro p1 p2
      run
      mac_exp p1, p2
endm
mac_exp macro add mask
      mac_flags = (mac_flags+add) & mask
endm
```

2.13 module

The *module* command is used to load and query external modules. A module is a special piece of software that can extend gpsim in some manner. LED's and switches are examples of modules. A module library is collection of modules.

Loading module libraries

```
module lib lib_name
```

The *lib* option is used to load a module library. Module libraries are system dependent shared libraries. I.e. on Windows they're DLL's and UNIX they're shared libraries. This means that either the libraries should reside in a path where the OS knows libraries exist or that the full path name must be specified along with the *lib_name*. gpsim provides a module library with a few modules:

```
gpsim> module lib libgpsim_modules
```

Displaying available modules

```
module list
```

The *list* option will display all of the modules that can be loaded. Here is an example of gpsim's built-in modules.

```
gpsim> module list
Module Libraries libgpsim_modules.so
  binary_indicator
  pullup
  pulldown
  usart
  parallel_interface
  switch
  and2
  or2
  xor2
  not
  led_7segments
  led
  PAL_video
  Encoder
```

Loading a specific module

```
module load module_type [module_name]
```

Once a library has been loaded, specific modules can be instantiated. The *module_type* is what's displayed by the *module list* command. The optional module name Here's an example

```
gpsim> module load led D1
```

Display loaded modules**Querying modules****2.14 node**

```
node [new_node1 new_node2 ...]
```

If no *new_node* is specified then all of the nodes that have been defined are displayed. If a *new_node* is specified then it will be

added to the node list. See the "attach" and "stimulus" commands to see how stimuli are added to the nodes.

examples:

```
node                // display the node list
node n1 n2 n3      // create and add 3 new nodes to the list
```

2.15 processor

```
processor [new_processor_type [new_processor_name]] | [list] | [dump]
```

The *processor* command is used to either define a new processor or to query one that has already been defined. Normally there's no need to explicitly define the processor since the symbol file already contains that information. The two exceptions are when a) the symbolic information is not available or b) you wish to override the processor specified in the symbol file. (See the *load* command on how the processor in a symbol file can be overridden.)

To see a list of the processors supported by gpsim, type '*processor list*'. To display the state of the I/O processor, type '*processor pins*'. For now, this will display the pin numbers and their current state.

```
examples:
processor          // Display the processors you've already defined.
processor list     // Display the list of processors supported.
processor pins     // Display the processor package and pin state
processor p16cr84 fred // Create a new processor.
processor p16c74 wilma // and another.
processor p16c65     // Create one with no name.
```

2.16 quit

Quit gpsim.

2.17 run

Start (or continue) simulation. The simulation will continue until the next break point is encountered.

2.18 step

```
step [over | n]
```

```
no arguments:  step one instruction.
numeric argument:  step a number of instructions
"over" argument:  step over the next instruction
```

2.19 symbol

```
symbol [symbol_name [symbol_type value]]
```

The *symbol* command is used to query and define symbols. If no options are specified, the whole symbol table is displayed. The creation of user defined symbols is limited at this time (see the online help for the current state of this command).

2.20 stimulus

```
stimulus [[type] options]
```

The *stimulus* command creates a signal that can be tied to a node or an attribute. If no options are specified then all currently defined stimuli are displayed.

Note that in most cases it is easier to create a stimulus file then to type the command by hand.

initial_state	state at the start and at the rollover
start_cycle	simulation cycle when the stimulus will begin
period	stimulus period
name	specifies the stimulus name

Here's an example of a stimulus that will generate two pulses and repeat this in 1000 cycles.

```
stimulus asynchronous_stimulus
# The initial state AND the state the stimulus is when
# it rolls over
initial_state 0
start_cycle 0
# the asynchronous stimulus will roll over in 'period'
# cycles. Delete this line if you don't want a roll over.
period 1000
{ 100, 1,
  200, 0,
```

```
    300, 1,  
    400, 0  
}  
# Give the stimulus a name:  
name two_pulse_repeat  
end
```

A stimulus can be queried by typing its name at the command line:

```
gpsim> two_pulse_repeat  
two_pulse_repeat attached to pulse_node  
Vth=0V Zth=250 ohms Cth=0 F nodeVoltage= 7.49998e-07V  
Driving=0 drivingState=0 drivenState=0 bitState=0  
states = 5  
    100 1  
    200 0  
    300 1  
    400 0  
    1000 0  
initial=0  
period=1000  
start_cycle=0  
Next break cycle=100
```

Even though this example uses 1's and 0's for the data, one can use integers, floating point numbers, or expressions instead. Integers are useful for supplying a stimulus to an attribute. Expressions are useful for abstracting the data. See Chapter 6 for more discussion and examples of stimuli.

2.21 stopwatch

2.22 trace

```
trace [dump_amount]  
trace will print out the most recent "dump_amount" traces.  
If no dump_amount is specified, then the entire trace buffer  
will be displayed.
```

2.23 version

```
version
```

Display gpsim's version. Note, this command will probably get replaced by an attribute with the same (or similar) name.

2.24 x

x [file_register] [new_value]

options:

file_register - ram location to be examined or modified.

new_value - the new value written to the file_register.

if no options are specified, then the entire contents
of the file registers will be displayed (dump).

Chapter 3

Graphical User Interface

gpsim also provides a graphical user interface that simplifies some of the drudgery associated with the cli. It's possible to open windows to view all the details about your debug environment. To get the most out of your debugging session, you'll want to assemble your code with gpasm (the gnupic assembler) and use the symbolic .cod files it produces.

3.1 Main window

3.1.1 Menus

File->Open .stc or .cod files.
File->Quit Quit gpsim
Windows->* Open/Close the windows.

3.1.2 Buttons

(These are also found as keyboard bindings in the source windows.)

Step	Step one instruction
Over	Step until pc is after next instruction
Finish	Run to return address
Run	Run continuously
Stop	Stop execution
Reset	Reset CPU

3.1.3 Simulation mode

This controls how gpsim simulates, and how the gui updates.

Never	Don't ever update the gui when simulating. This is the fastest mode. You'll have to stop simulation by pressing Ctrl-C in the command line interface.
x cycles	Update the gui every x cycles simulated.
every cycle	Update the gui every cycle. (you see everything, if you have filled up on coffee :-)
x ms animate	Here you can slow down simulation with a delay between every cycle.
realtime	This will make gpsim try to synchronize simulation speed with wall clock time.

3.2 Source Browsers

gpsim provides two views of your source: `.asm` and `.obj` browsers. The `.asm` browser is a color coded display of your pic source.

3.2.1 .asm Browser

When a `.cod` file with source is loaded, there should be something in this display. (TODO: add section about high level debugging).

There is an area to the left of the source, where symbols representing the program counter, breakpoints, etc are displayed. Double clicking in this area toggles breakpoints. You can drag these symbols up or down in order to move them and change the PC or move a breakpoint.

A right button click on the source pops up a menu with six items (the word 'here' in some menu items denote the line in source the mouse pointer was on when right mouse button was clicked.):

Menu item	Description
Find PC	This menu item will find the PC and changed page tab and scroll the source view to the current PC.
Run here	This sets a breakpoint 'here' and starts running until a breakpoint is hit.
Move PC here	This simply changes PC to the address that line 'here' in source has.
Breakpoint here	Set a breakpoint 'here'.

Profile start here	Set a start marker for routine profiling here.
Profile stop here	Set a stop marker. (See the section for the profiling window.)
Select symbol.	This menu item is only available when some text is selected in the text widget. What it does is search the list of symbols for the selected word, and if it is found it is selected in the symbol window. Depending of type of symbol other things are also done, the same thing as when selecting a symbol in the symbol window: <ul style="list-style-type: none"> • If it is an address, then the opcode and source views display the address. • If it's a register, the register viewer selects the cell. • If it's a constant, address, register or ioport, it is selected in the symbol window.
Find text	This opens up a search dialog. Every time you hit the 'Find' button, the current notebook page is found and the source in that page is used.
Settings	A dialog with which you can change the fonts used.
Controls	A submenu containing the simulation commands. (these are also found as keyboard bindings (recommended), or in the main window.)

These are the keyboard bindings:

Key	command
s,S,F7	Step one instruction.
o,O,F8	Step over instruction
r,R,F9	Run continuously.
Escape	Stop simulation.
f,F	Run to return address

3.2.2 Opcode view - the .obj Browser

This window has two tabs. One with each memory cell on one line and information about address, hexadecimal value and decoded instruction (i.e. disassembly), and one with the program memory displayed with sixteen memory cells per row and a configurable ascii column.

The Assembly tab you can:

- Double click on a line to toggle breakpoints.
- Use the same keyboard commands as the in the source browser.
- Right click to get a menu where you can change the fonts.

The Opcode tab.

Here the program memory is ordered as columns of sixteen memory cells per column and as many row as needed to contain all memory.

The seventeenth column contains an ASCII representation of the program memory. You can configure this column to use one of three different modes:

- One byte per cell
- Two bytes per cell, MSB first.
- Two bytes per cell, LSB first.

You can change fonts with the menu item 'Settings'.

You can set breakpoints on one or more (drag the mouse to select more cells) addresses with the right click menu.

3.3 Register views

There are two similar register windows. One for the RAM and one for the EEPROM data, when available.

Here you see all registers in the current processor. Clicking on a cell displays it's name and value above the sheet of registers. You can change values by entering it in the entry (or in the spreadsheet cell).

The following things can be done on one register, or a range of registers. (Selecting a range of registers is done by holding down left mouse button, moving cursor, and releasing button.)

- Set and clear breakpoints. Use the right mousebutton menu to pop up a menu where you can select set read, write, read value and write value breakpoints. You can also "clear breakpoints", notice the s in "clear breakpoints", every breakpoint on the registers are cleared.
- Set and clear logging of registers. You can log reads, writes, reads/writes of specific values and to bits selected by a specified mask. You can select a different file name with 'set log filename...'. Default is "gpsim.log". You can choose LXT or ASCII format. LXT can be read with the program gtkwave. ASCII is default.

- Copy cells. You copy cells by dragging the border of the selected cell(s).
- Fill cells. Move mouse to lower right corner of the frame of the selected cell(s), and drag it. The one cell's contents will be copied to the other cells.
- Watch them. Select the "Add Watch" menu item.

The cells have different background colors depending on if they represent:

- File Register (e.g. RAM): light cyan.
- Special Function Registers (e.g. STATUS, TMR0): dark cyan
- aliased register (e.g. the INDF located at address 0x80 is the same as the one located at address 0x00): gray
- invalid register: black. If all sixteen registers in a row are invalid, then the row is not shown.
- a register with one or more breakpoints: red. Logged registers are also red.

gpsim dynamically updates the registers as the simulation proceeds. Registers that change value between updates of the window during simulation are highlighted with a blue foreground color.

The menu also has a 'settings' item where you can change the font used.

3.4 Symbol view

This window, as its name suggests, displays symbols. All of the special function registers will have entries in the symbol viewer. If you're using .cod files then you'll additionally have file registers (that are defined in cblocks), equates, and address labels.

You can filter out some symbol types using the buttons in the top of the window, and you can sort the rows by clicking on the column buttons (the ones reading 'symbol', 'type' and 'address').

You can add the symbol to the watch window by right-clicking and selecting the "Add to watch window" menu item. This will add the ram register with address equal to the symbols value to the watch window.

The symbol viewer is linked to the other windows. For example, if you click on a symbol and:

- If it is an address, then the opcode and source views display the address.
- If it's a register, the register viewer selects the cell.

3.5 Watch view

This is not a output-only window as the name suggests (change name?). You can both view and change data. Double-clicking on a bit toggles the bit. You add variables here by marking them in a register viewer and select “Add watch” from menu. The right-click menu has the following items:

- Remove watch
- Set register value
- Clear Breakpoints
- Set break on read
- Set break on write
- Set break on read value
- Set break on write value
- Columns...

“Columns...” opens up a window where you can select which of the following data to display:

- BP
- Type
- Name
- Address
- Dec
- Hex
- Bx (bits of word)

You can sort the list of watches by clicking on the column buttons. Clicking twice sorts the list backwards.

3.6 Stack viewer

This window displays current stack. Selecting an entry makes the code windows display the return address. Double clicking sets a breakpoint on the return address.

3.7 Breadboard

Here you can create/modify and examine the environment around the pic. Pins are displayed as an arrow. The direction of the arrow indicates if its an input or output pin. The color of the arrow indicates its state (green=low, red=high).

You can't instantiate pic processors from here, you'll have to do that from the command line, or from a .stc file.

You can create nodes by clicking on the "new node" button. (A node is 'a piece of wire' to which you can connect stimulus.) You can see the list of created nodes under the "nodes" item in the upper-left tree widget.

You can create connections to nodes by clicking on a pin, and then clicking on the button "Connect stimulus to node". This will bring up a list of nodes. Choose one by double-clicking on the one you like.

If you click on a pin that is already connected to a node, then you'll see the node and its connections in the lower left part of the window. You can disconnect a stimulus by clicking on it and pressing the "remove stimulus" button.

When you want to add a module to the simulation, you first have to specify the library which contains the module you want. Click on the "add library" button and enter the library name (e.g. "libgpsim_modules.so"). Now you can click the "add module" button. Select the module you want from the list by double-clicking on it. Enter a name for the module (this has to be unique, and not used before). You now have to position the module. Move the mouse pointer to where you'd like the module, and left-click.

If you middle-click on a pin, you'll see how the pin is connected. Press the "trace all" to see all at

once, and "clear traces" to remove all (you'll only remove the graphical trace, not the connection!). If the tracing doesn't work, try moving the packages so that there are more space around the pins.

When you are done, you can save by clicking the "save configuration" button. You can then load this file from the command line like this (assuming the .cod file with your source is called "mycode.cod", and the file you just saved was called "mynets.stc"):

```
gpsim -s mycode.cod -c mynets.stc
```

You can't load only the .stc file since this doesn't contain the processor type and code. You can create (with an editor) your own .stc file (e.g. my_project.stc) and in that file put a command "load c mynets.stc" after you have loaded the .cod file. You then only have to load this file (gpsim -c my_project.stc).

3.8 Trace viewer

This window shows the trace of instructions executed. See 5.

3.9 Profile viewer

This window show execution count for program memory addresses. The profile window must be opened before starting simulation, because the tracing is not enabled by default.

Instruction profile

This shows the number of times each instruction are executed.

Instruction range profile

Here you can group ranges of instruction into one entry.

The right click menu contains:

Remove range	Remove an entry.
Add range...	Open a dialog from where you can add a range of instructions as an entry.
Add all labels	Add all code labels as ranges.
Snapshot to plot	Open a window containing a graph of the data. From this new window you can also save (postscript) or print it.

Register profile

This shows the number of reads or writes the simulator does on register.

Routine profile

Here you can see statistics about execution time for a selected routine. You mark the entry and exit points from the source browser (profile start/stop). If the routine you want to measure have multiple entry and/or exit points, then you have to put a marker on every entry point as well as (and especially) every exit point. Otherwise you will get bad data.

When you have done that, gpsim will (as simulation goes by) store the execution times of that routine and calculate min/max/average/etc. You can also use the menu item 'Plot distribution' to open a window displaying a histogram of the data. From this new window you can also save (in postscript) or print it.

You can also measure call period by switching the 'entry' and 'exit' points. If also want the time from reset (or some equal point) to the first 'entry', then you must also put an 'entry' point there.

3.10 Stopwatch

The stopwatch window shows a cycle counter and a re-settable counter. The cycle counter is the same as the one in the register window. It basically counts instructions.

The other counter counts at the same rate as the cycle counter, but can be cleared by clicking the "clear" button (or preset by entering a number in the entry box).

The up/down indicator denotes the direction the counter counts.

The rollover value specifies the range the cycle counter can be in (a modulo counter). For example, if the rollover value is specified to be 0x42, then whenever the resettable counter reaches 0x42 it will rollover to zero. If the counter is counting down, then when it reaches 0 the next state will be 0x41. If you don't want it like this, then set the rollover value to something large.

Chapter 4

Controlling the Flow: Break Points

One of gpsim's strong features is the flexibility provided with break points. Most simulators are limited to execution type break points.

If you want to set break points on registers, on execution cycles, invalid program locations, stack over flows, etc. then you're usually forced to debug your code with an ICE.

4.1 Execution Break Points

An execution break point is one that will halt a running program when the program memory address at which it is set is encountered. For example, if you were debugging a mid-ranged PIC and wished to stop execution when ever an interrupt occurs, you could set a break point at program memory address 0x04:

```
gpsim> break e 4
```

(To be more precise, an interrupt doesn't have to occur for this break point to be encountered - errant code could have branched here too).

The break point occurs BEFORE the instruction executes. Other simulators such as MPLAB break after the instruction executes. In many cases this distinction is insignificant. However, if the break is set on a 'goto' or 'call' instruction, then it's convenient to stop before the branch occurs. This way it's easy to determine from where a brance occurred.

4.1.1 Invalid Instruction Break Points

gpsim automatically will halt execution if a program attempts to venture beyond its bounds. Program memory locations that are not defined by your source code will be

initialized with an 'Invalid Instruction'. These are quite visible when you disassemble the program.

4.2 Register Break Points

gpsim provides the ability to break whenever a register accessed, either read or written or both. Furthermore, it's possible to break whenever a specific value is written to or read from a register.

4.3 Cycle Break Points

Cycle break points allow the program to be halted at a specific instruction cycle. Suppose you have a 20 Mhz pic and want to break after one second of simulation time. You could set a break at the 5 millionth instruction cycle.¹

¹There are 4 clock cycles per instruction. Also, a future feature of gpsim will provide you with the ability to set break points in terms of seconds.

Chapter 5

Trace: What has happen?

Inspecting the current state of your program is sometimes insufficient to determine the cause of a bug. Often times it's useful to know the conditions that led up to the current state. gpsim provides a history or trace of everything that occurs - whether you want it or not - to help you diagnose these otherwise difficult to analyze bugs.

What's traced	notes
program counter	addresses executed
instructions	opcode
register read	value and location
register write	value and location
cycle counter	current value
skipped instructions	addresses skipped
status register	during implicit modification
interrupts	
break points	type
resets	type

The 'trace' command will dump the contents of the trace buffer.

A large circular buffer (whose size is hard coded) stores the information for the trace buffer. When it fills, it will wrap around and write over the old history. The contents of the trace buffer are parsed into frames, where one frame corresponds to a simulation cycle.

Here's an example of a trace output:

```
gpsim> trace
0x000000000000026F6 p18f452 0x001C 0x1003 iorwf reg3,w,0
```

```

    Read: 0x00 from reg3(0x0003)
    Wrote: 0xE7 to W(0x0FE8) was 0xE7
    Wrote: 0x18 to status(0x0FD8) was 0x18
0x00000000000026F7 p18f452 0x001E 0xE1F4 bnz    $-0x16    ;(0x8)
0x00000000000026F8 p18f452 0x0008 0x3E00 incfsz reg,f,0
    Read: 0xE4 from reg(0x0000)
    Wrote: 0xE5 to reg(0x0000) was 0xE4
0x00000000000026F9 p18f452 0x000A 0xD004 bra    $+0xa    ;(0x00014) 0x000000
0x00000000000026FB p18f452 0x0016 0x5000 movf   reg,w,0
    Read: 0xE5 from reg(0x0000)
    Wrote: 0xE5 to W(0x0FE8) was 0xE7
    Wrote: 0x18 to status(0x0FD8) was 0x18
0x00000000000026FC p18f452 0x0018 0x1001 iorwf  reg1,w,0
    Read: 0x03 from reg1(0x0001)
    Wrote: 0xE7 to W(0x0FE8) was 0xE5
    Wrote: 0x18 to status(0x0FD8) was 0x18

```

Each trace frame begins with a new simulation cycle. Typically this will include a simulated instruction. Here's each of the fields:

```

64-bit simulation cycle  processor    PC    opcode    instruction
0x00000000000026F6     p18f452  0x001C  0x1003  iorwf reg3,w,0

```

Other events that occur during the trace frame are indented. Typically these will be register read or write traces. The read traces show the value read. Write traces show the value written and the value that was previously in the register.

Chapter 6

Simulating the Real World: Stimuli

Stimuli are extremely useful, if not necessary, for simulations. They provides a means for simulating interactions with the real world.

The gpsim stimuli capability is designed to be accurate, efficient and flexible. The models for the PIC's I/O pins mimic the real devices. For example, the open collector output on port A of an PIC16C84 can only drive low. Multiple I/O pins may tied to one another so that the open collector on port A can get a pull up resistor from port B. The overhead for stimuli only occurs when a stimulus changes states. In other words, stimuli are not polled to determine their state.

Analog stimuli are also available. It's possible to create voltage references and sources to simulate almost any kind of real world thing. For example, it's possible to combine two analog stimuli together to create signals like DTMF tones.

6.1 How They Work

In the simplest case, a stimulus acts a source for an I/O pin on a pic. For example, you may want to simulate a clock and measure its period using TMR0. In this case, the stimulus is the source and the TMR0 input pin on the pic is the load. In gpsim you would create a stimulus for the clock using the stimulus command and connect it to the I/O pin using the node command.

In general, you can have several 'sources' and several 'loads' that are interconnected with nodes¹. A good analogy is a spice circuit. The spice netlist corresponds to a node-list in gpsim and the spice elements correspond to the stimuli sources and loads. This general approach makes it possible to create a variety of simulation environments. Here's a list of different ways in which stimuli may be connected:

¹Although, gpsim is currently limited to 'one-port' devices. In other words, it is assumed that ground serves as a common reference for the sources and the loads.

1. Stimulus connected to one I/O pin
2. Stimulus connected to several I/O pins
3. Several stimuli connected to one I/O pin
4. Several stimuli connected to several I/O pins
5. I/O pins connected to I/O pins

The general technique for implementing stimuli is as follows:

1. Define the stimulus or stimuli.
2. Define a node.
3. Attach the stimuli to the node.

More often than not, the stimulus definition will reside in a file.

6.1.1 Contention among stimuli

One of the problems with this nodal approach to modeling stimuli is that it's possible for contention to exist. For example, if two I/O pins are connected to one another and driving in the opposite directions, there will be contention. `gpsim` resolves contention with attribute summing. Each stimulus - even if it's an input - has an effect on the node. This effect is given a weight. When a node is updated, `gpsim` will simply add the weights of all the stimuli together and assign that numeric value to the node. A weight value of zero corresponds to no load. A large positive weight is used by a stimulus to drive the node positive, while a large negative weight is used to drive it negative.

Attribute summing is useful for pull up resistors. In the port A open collector / port B weak pull-up connection example, `gpsim` assigns a relatively small weight to the pull up resistor and a large negative weight to the open collector if it is active or no weight if it's not driving. Capacitive effects (which are not currently supported) can be simulated with dynamically changing weight values.

6.2 I/O Pins

`gpsim` models I/O pins as stimuli. Thus anywhere a stimulus is used, an I/O pin may be substituted. For example, you may want to tie two I/O pins to one another; like a port B pull up resistor to a port A open collector. `gpsim` automatically creates the I/O pin stimuli whenever a processor is created. All you need to do is to specify a node and then attach the stimuli to it. The names of these stimuli are formed by concatenating the port name with the bit position of the I/O pin. For example, bit 3 in port B is called `portb3`.

Here's a list of the types of I/O pin stimuli that are supported:

I/O Pin Type	Function
INPUT_ONLY	Only accepts input (like MCLR)
BI_DIRECTIONAL	Can be a source or a load (most I/O pins)
BI_DIRECTIONAL_PU	PU=Pullup resistor (PORTB)
OPEN_COLLECTOR	Can only drive low (RA4 on c84)

There is no special pin type for analog I/O pins. All pic analog inputs are multiplexed with digital inputs. The I/O pin definition will always be for the digital input. gpsim automatically knows when I/O pin is analog input.

6.3 Asynchronous Stimuli

Asynchronous stimuli are analog or digital stimuli that can change states at any given instant (limited to the resolution of the cycle counter). They can be defined to be repetitive too.

parameter	function
start_cycle	The # of cycles before the stimulus starts
cycles[]	An array of cycle #'s
data[]	Stimulus state for a cycle
period	The # of cycles for one period
initial_state	The initial state before data[0]

When the stimulus is first initialized, it will be driven to the 'initial state' and will remain there until the cpu's instruction cycle counter matches the specified 'start' cycle. After that, the two arrays 'cycles[]' and 'data[]' define the stimulus' outputs. The size of the arrays are the same and correspond to the number of events that are to be created. So the event number, if you will, serves as the index into these arrays. The 'cycles[]' array define when the events occur while the 'data[]' array defines the states the stimulus will enter. The 'cycles[]' are measured with respect to the 'start' cycle. The asynchronous stimulus can be made periodic by specifying the number of cycles in the 'period' parameter.

Here's an example that generates three pulses and then repeats:

```
stimulus asynchronous_stimulus # or we could've used asy
# The initial state AND the state the stimulus is when
# it rolls over
initial_state 1
# all times are with respect to the cpu's cycle counter
start_cycle 100
# the asynchronous stimulus will roll over in 'period'
```

```
# cycles. Delete this line if you don't want a roll over.
period 5000
# Now the cycles at which stimulus changes states are
# specified. The initial cycle was specified above. So
# the first cycle specified below will toggle this state.
# In this example, the stimulus will start high.
# At cycle 100 the stimulus 'begins'. However nothing happens
# until cycle 200+100.
{ 200, 0,
  300, 1,
  400, 0,
  600, 1,
  1000, 0,
  3000, 1 }
# Give the stimulus a name:
name asy_test
# Finally, tell the command line interface that we're done
# with the stimulus
end
```

6.3.1 Analog Asynchronous Stimuli

Analog Asynchronous Stimuli are identical to Synchronous Stimuli except the data points are floating point numbers.

Chapter 7

Modules

gpsim has been designed to debug microprocessors. However, microprocessors are always a part of a system. And invariably, the bugs one often encounters are those that are a result of interfacing with a system. Modules provide users with a way to extend gpsim and simulate a system. For example, the *system* may be a processor with a few pull up resistors and switches or it may be a processor and an LCD display. gpsim provides a few modules that one may use either for debugging or as templates for creating new modules.

7.1 gpsim Modules

gpsim provides the following modules:

binary_indicator	
pullup	
pulldown	
usart	
parallel_interface	
switch	
and2	
or2	
xor2	
not	
led_7segments	
led	
PAL_video	
Encoder	

7.2 Writing new modules

A module is a library of code. On Windows the library is a .DLL and Unix a shared library. There are few details that module must adhere to, but in general the module has full access to gsim's API.

Chapter 8

Symbolic Debugging

gpsim maintains a symbol table.

<write me>

Chapter 9

Macros

<write me>

Chapter 10

Hex Files

The target code simulated by gpsim can be supplied by a hex file, or more specifically an Intel Hex file. gpsim accepts the format of hex provided by gpasm and mpasm. The hex file does not provide any symbolic information. It's recommended that hex files only be used if 1) you suspect there's a problem with the way .cod files are generated by your assembler or compiler OR 2) your assembler or compiler doesn't generate .cod files. Also, you must supply a processor when loading hex files. See the load command.

Chapter 11

The ICD

gpsim supports (partly) the first version of the ICD (as opposed to ICD2 (the round hockey-puck shaped one)).

Special configuration of the code

Read the MPLAB ICD USER's GUIDE.

Here's the short version:

- disable at least: brown out detection, low voltage programming and all code protection. It is probably good to turn of the watchdog too. see the MPLAB ICD USER's GUIDE for more information.
- have a NOP as the first instruction.
- Don't touch RB6 or RB7.
- Don't use the last stack level.
- Don't use these registers and program words:

Processor	Register	Program
-870/1/2	0x70, 0xBB-0xBF	0x6E0-0x7FF
-873/4	0x6D, 0x1fD, 0xEB-0xF0, 0x1Eb-0x1F0	0xEE0-0xFFF
-876/7	0x70, 0x1Eb-0x1Ef	0x1F00-0x1FFF

icdprog

Download and install icdprog.

Use icdprog to program the target with the hex file (*icdprog mycode.hex*).

ICD usage

Start gpsim like this:

```
gpsim -d /dev/ttyS0 -s mycode.cod
```

, assuming the ICD is connected to the first serial port.

Now you can type 'icd' to see some information:

```
**gpsim> icd
ICD version "2.31.00" was found.
Target controller is 16F877 rev 13.
Vdd: 5.2 Vpp: 13.3
Debug module is present
```

2.31 is the firmware version. I have only tried this particular version...

You can step, reset, run, halt, set the breakpoint and read file registers. It works both from the gui and the cli.

ICD TODO

- MPLAB has a setting for target cpu frequency, I have only tried with a 20MHz crystal, so there may be adjustments to be made to the serial port timeout settings in gpsim.
- The source, disassembly, watch, symbol and RAM windows works. And the rest doesn't. I guess the breadboard should be able to work at least for the pic, but it doesn't.
- eeprom support
- modifying data
- Fix the UI to give more feedback about what's happening during long delays.
- Better error detection. gpsim doesn't always see that the target is not functional.

Chapter 12

Theory of Operation

This section is only provided for those who may be interested in how gpsim operates. The information in here is 'mostly' accurate. However, as gpsim evolves so do the details of the theory of operation. Use the information provided here as a high level introduction and use the (well commented :) source to learn the details.

12.1 Background

gpsim is written mostly in C++. Why? Well the main reason is to easily implement a hierarchical model of a pic. If you think about a microcontroller, it's really easy to modularize the various components. C++ lends itself well to this conceptualization. Furthermore Microchip, like other microcontroller manufacturers, has created families of devices that are quite similar to one another. Again, the C++ provides 'inheritance' that allows the relationships to be shared among the various models of pics.

12.2 Instructions

There's a base class for the 14-bit instructions (I plan to go one step further and create a base class from which all pic instructions can be derived). It primarily serves two purposes: storage that is common for each instruction and a means for generically accessing virtual functions. The common information consists of a name - or more specifically the instruction mnemonic, the opcode, and a pointer to the processor owning the instruction. Some of the virtual functions are 'execute' and 'name'. As the hex file is decoded, instances of the instructions are created and stored in an array called `program_memory`. The index into this array is the address at which the instruction resides. To execute an instruction the following code sequence is invoked:

```
program_memory[pc.value]->execute();
```

which says, get the instruction at the current program counter (`pc.value`) and invoke via the virtual function `execute()`. This approach allows execution break points to be easily set. A special break point instruction can replace the one residing in the program memory array. When `'execute'` is called the break point can be invoked.

12.3 General File Registers

A file register is simulated by the `'file_register'` class. There is one instance of a `'file_register'` object for each file register in the PIC. All of the registers are collected together into an array called `'registers'` which is indexed by the registers' corresponding PIC addresses. The array is linear and not banked like it is in the PIC. (Banking is handled during the simulation.)

12.4 Special File Registers

Special file registers are all of the other registers that are not general file registers. This includes the core registers like `status` and `option` and also the peripheral registers like `eadr` for the eeprom. The special file registers are derived from the general file registers and are also stored in the `'registers'` array. There is one instance for each register - even if the register is accessible in more than one bank. So for example, there's only one instance for the `'status'` register, however it may be accessed through the `'registers'` array in more than one place.

All file registers are accessed by the virtual functions `'put'` and `'get'`. This is done for two main reasons. First, it conveniently encapsulates the breakpoint overhead (for register breakpoints) in the file register and not in the instruction. Second, and more important, it allows derived classes to implement the `put` and `get` more specifically. For example, a `'put'` to the `indf` register is a whole lot different than a `put` to the `intcon` register. In each case, the `'put'` initiates an action beyond simply storing a byte of data in an array. It also allows the following code sequence to be easily implemented:

```
movlw    trisa    ;Get the address of tris
movwf    fsr
movf     indf,w   ;Read trisa indirectly
```

12.5 Example of an instruction

Here's an example of the code for the `movf` instruction that illustrates what has been discussed above. Somewhere in `gpsim` the code sequence:

```
program_memory[pc.value]->execute();
```

is executed. Let's say that the pc is pointing to a movf instruction. The `->execute()` virtual function will invoke `MOVF::execute`. I've added extra comments (that aren't in the main code) to illustrate in detail what's happening.

```
void MOVF::execute(void)
{
    unsigned int source_value;

    // All instructions are 'traced' (discussed below). It's sufficient
    //to only store the opcode. However, even this may be unnecessary since
    //the program counter is also traced. Expect this to disappear in the
    //future...
    trace.instruction(opcode);

    // 'source' is a pointer to a 'file_register' object. It is initialized
    //by reading the 'registers' array. Note that the index depends on the
    //'rp' bits (actually just one bit) in the status register. Time is
    // saved by caching rp as opposed to decoding the status register.
    source = cpu->registers[cpu->rp | opcode&REG_IN_INSTRUCTION_MASK];

    // We have no idea which register we are trying to access and how it
    //should be accessed or if there's a breakpoint set on it. No problem,
    //the virtual function 'get' will resolve all of those details
    // and 'do the right thing'.
    source_value = source->get();

    // If the destination is W, then the constructor has already initialized
    //'destination'. Otherwise the destination and source are the same.
    if(opcode&DESTINATION_MASK)
        destination = source;    // Result goes to source

    // Write the source value to the destination. Again, we have no idea
    // where the destination may be or
    // or how the data should be written there.
    destination->put(source_value);

    // The movf instruction will set Z (zero) bit in the status register
    //if the source value was zero.
    cpu->status.put_Z(0==source_value);

    // Finally, advance the pc by one.
    cpu->pc.increment();
}
```

12.6 Trace

Everything that is simulated is traced - *all* of the time. The trace buffer is one huge circular buffer of integers. Information is or'ed with a trace token and then is stored in the trace buffer. No attempt is made to associate the items in the trace buffer while the simulator is simulating a PIC. Thus, if you look at the raw buffer you'll see stuff like: cycle counter = ..., opcode fetch = ..., register read = ..., register write = ..., etc. However, this information is post processed to ascertain what happened and when it happened. It's also possible to use this information to undo the simulation, or in other words you can step backwards. I don't have this implemented yet though.

12.7 Breakpoints

Breakpoints fall into three categories: execution, register, and cycle.

Execution:

For execution breakpoints a special instruction appropriately called 'Breakpoint_Instruction' is created and placed into the program memory array at the location the break point is desired. The original instruction is saved in the newly created breakpoint instruction. When the break point is cleared, the original instruction is fetched from the break point instruction and placed back into the program memory array.

Note that this scheme has zero overhead. The simulation is only affected when the breakpoint is encountered.

Register:

There are at least four different breakpoint types that can be set on a register: read any value, write any value, read a specific value, or write a specific value. Like the execution breakpoints, there are special breakpoint registers that replace a register object. So when the user sets a write breakpoint at register 0x20 for example, a new breakpoint object is created and insert into the file register array at location 0x20. When the simulator attempts to access register location 0x20, the breakpoint object will be accessed instead.

Note that this scheme too has zero overhead, except when a breakpoint is encountered.

Cycle:

Cycle breakpoints allow gpsim to alter execution at a specific instruction cycle. This is useful for running your simulation for a very specific amount of time. Internally, gpsim makes extensive use of the cycle breakpoints. For example, the TMR0 object can be programmed to generate a periodic cycle break point.

Cycle break points are implemented with a sorted doubly-linked list. The linked list contains two pieces of information (besides the links): the cycle at which the break is to occur and the call back function¹ that's to be invoked when the cycle does occur. The break logic is extremely simple. Whenever the cycle counter is advanced (that is, incremented), it's compared to the next desired cycle break point. If there's NO match, then we're done. So the overhead for cycle breaks is the time required to implement a comparison. If there IS a match, then the call back function associated with this break point is invoked and the next break point in the doubly-linked list serves as the reference for the next cycle break.

¹A call back function is a pointer to a function. In this context, gpsim is given a pointer to the function that's to be invoked (called) whenever a cycle break occurs.

COPYING

The document is part of gpsim.

gpsim is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

gpsim is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with gpsim; see the file COPYING. If not, write to the Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Index

attach, 9

break, 9, 10

bus, 9

clear, 9, 11

disassemble, 9, 12

dump, 9, 12

echo, 12

frequency, 9, 12

GNU, 50

help, 9, 13

icd, 9, 13

instructions, 45

License, 50

list, 9, 13

load, 9, 13

log, 9

macros, 14

module, 9, 15

NO WARRANTY, 50

node, 9, 16

processor, 9, 17

quit, 9, 17

registers, 46

run, 9, 17

set, 9

step, 9, 18

Stimulus, 34

stimulus, 9, 18

stopwatch, 9, 19

symbol, 9, 18

trace, 9, 19

version, 9

x, 9, 20