gpsim

$Date: 2006-11-09 08:27:47 -0800 (Thu, 09 Nov 2006) $

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 12

2.4 disassemble 12

2.5 dump 12

2.6 echo 12

2.7 frequency 13

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 17

2.15 processor 17

2.16 quit 17

2.17 run 18

2.18 step 18

2.19 symbol 18
2.20 stimulus 18
2.21 stopwatch 19
2.22 trace 20
2.23 version 20
2.24 x 20

3 Graphical User Interface **22**
3.1 Main window 22
3.1.1 Menus 22
3.1.2 Buttons 22
3.1.3 Simulation mode 23
3.2 Source Browsers 23
3.2.1 .asm Browser 23
3.2.2 Opcode view - the .obj Browser 24
3.3 Register views 25
3.4 Symbol view 26
3.5 Watch view 27
3.6 Stack viewer 27
3.7 Breadboard 28
3.8 Trace viewer 28
3.9 Profile viewer 29
3.10 Stopwatch 30
3.11 Scope Window 30

4 Scripting and Configuring **31**
4.1 Embedded Commands 31
4.1.1 .sim macro 31
4.1.2 .command macro 32
4.1.3 .assert macro 32
4.2 Sockets 33

5 Assertions and Extended Breakpoints **34**
5.1 Assertions and Embedded Simulation commands 35

6 Trace and Log: What has happen? **36**

7 Simulating the Real World: Stimuli **39**
7.1 How They Work 39
7.1.1 Contention among stimuli 40
7.2 I/O Pins 40
7.3 Asynchronous Stimuli 41
7.3.1 Analog Asynchronous Stimuli 42
7.4 Extended Stimuli 42

8 Modules **43**
8.1 gpsim Modules 44
8.1.1 USART 44
8.1.2 Logic 45
8.1.3 I2C EEPROM 46
8.1.4 Switches & Resistors 46
8.1.5 Voltage Sources, Resistors, and Capacitors 47
8.2 Third Party Modules 47
8.2.1 Character LCD - HD44780 47
8.2.2 Graphic LCD - SED1530 47
8.3 Writing new modules 47

9 Symbolic Debugging **50**

10 Macros **51**

11 Hex Files **52**

12 The ICD- Not Supported in versions 0.21.0 and later **53**

13 Examples **55**

14 Regression Tests **56**

15 Theory of Operation **57**
15.1 Background 57
15.2 Instructions 57
15.3 General File Registers 58
15.4 Special File Registers 58
15.5 Example of an instruction 58
15.6 Trace 60
15.7 Breakpoints 60

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 created in a manner that's consistent with much of the other open source software:

command	description
tar -xvzf gpsim-x.y.z.tar.gz	expand the compressed tar file
./configure	Create a 'makefile' unique to your system
make	compile gpsim
make install	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. Snap shots 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/Progam 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 Fedora, Ubuntu, etc. 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 .cod[1] files that gputils produces. The .cod files are in the same format as the .cod files MPASM[2] produces.

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

[2]MPASM 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	(deprecated) 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. See chapter 5 for more examples of breakpoints.

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 auxillary 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 intend mainly for module writers to provide a mechanism for allowing the user to control the module.

Cycle counter Breakpoints

```
break c cycle_number
```

The cycle counter is gpsim's time keeper. It increments once every instruction cycle. The 'c' option to the break command allows a break point to be set at a particular value of the cycle counter.

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]

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.

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

2.7 frequency

This command sets the clock frequency. By default gpsim uses 20 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 [processortype] file
```

gpsim will automatically determine if the file is a .hex or .cod file. The optional processortype 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 enviroment 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 (see section 8 for more information about gpsim 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
   pullup
   pulldown
   usart
   switch
   and2
   or2
   xor2
   not
   led_7segments
   led
   pulsegen
   Encoder
   TTL377
```

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 specifies what the instance is called. Here's an example

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

Display loaded modules

Querying modules

2.14 node

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

The *node* command defines or queries “nodes”, used to connect external signals to the simulated PIC. 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

Execute a single instruction, or a specified number of instructions.

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

With no arguments, the step command executes one instruction of the PIC code. If a numeric argument is given, this specifies a fixed number of instructions to simulate. The specific word "over" as an argument to step tells gpsim to run everything involved in the current instruction. This would normally be used on a CALL instruction, in which case the whole subroutine runs and the simulation stops after it returns.

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 cyle 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 7 for more discussion and examples of stimuli.

2.21 stopwatch[2]

```
A timer for monitoring and controlling the simulation.
```

[2]The stopwatch is really a collection of attributes and not a command. But the behavior is so similar to a command that it has been included here.

```
The units are in simulation cycles.
  stopwatch.rollover - specifies rollover value.
  stopwatch.direction - specifies count direction.
  stopwatch.enable - enables counting if true.
```

Without any options, *stopwatch* will display the contents of the stopwatch timer. *stopwatch* is writable, so you may initialize it to whatever value you like. The behavior of the timer may be manipulated via the three attributes. The *.rollover* attribute is the number of cycles at which the stopwatch timer rolls over. The *.direction* and *.enable* attributes are boolean types. When true, the *.direction* attribute will instruction the stopwatch to count up.

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

The *x* command is deprecated. It's former use was to examine and modify memory. The preferred way to do this now is with expressions. The help for *x* now indicates this:

```
x examine command -- deprecated
  Instead of the using a special command to examine and modify
  variables, it's possible to directly access them using gpsim's
  expression parsing. For example, to examine a variable:
gpsim> my_variable
my_variable [0x27] = 0x00 = 0b00000000
  To modify a variable
gpsim> my_variable = 10
  It's also possible to assign the value of register to another
gpsim> my_variable = porta
  Or to assign the results of an expression:
```

```
gpsim> my_variable = (porta ^ portc) & 0x0f
```

Chapter 3

Graphical User Interface

FIXME: We could use a few illustrations here!

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: • 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 continously.
Escape	Stop simulation.
f,F	Run to return address
1..9	Step n instructions

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.

Your 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 6.

3.9 Profile viewer

This window show execution count for program memory addresses. The profile window must be opened before starting simulation, because the traceing 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 disalog 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. Othervise 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 is like this, then set the rollover value to something large.

3.11 Scope Window

FIXME: The scope window still needs some work...

The Scope Window graphs I/O pin states. It is similar to an oscilloscope or logic analyzer. It can be controlled either from the command line or from the gui. Currently only the digital state of I/O pins are supported.

Chapter 4

Scripting and Configuring

gpsim does not have a native scripting language per se. However it is possible to place gpsim commands into a file and load them later. This is useful for loading modules and stimuli and connecting various devices together. By convention, gpsim's configuration files have the extension *.stc*, for *st*artup *c*onfiguration.

4.1 Embedded Commands

If you're using gputils, it is possible to embed configuration commands directly into your PIC assembly source. The gputils supplied include file *coff.inc* contains several macros that embed simulation command into a COFF and COD files.

4.1.1 .sim macro

```
; Simulator Command
.sim macro x
  .direct "e", x
  endm
```

The *.sim* macro allows gpsim configuration commands to be embedded in the PIC source. While gpsim loads a .cod file, the commands in the *.sim* macros are collected. After the .cod file is loaded, the commands are redirected to gpsim's command line interpreter in the order they were received.

Here's an example of switch module being loaded and configured:

```
;# Module libraries:
.sim "module library libgpsim_modules"
.sim "module load switch SW1"
.sim "SW1.state=false"
```

```
.sim "SW1.xpos = 216.0"
.sim "SW1.ypos = 156.0"
.sim "SW1.Ropen = 1.0e8"
```

This loads gpsim's module library, instantiates a switch module, and configures the switch's attributes.

4.1.2 .command macro

```
.command macro x
  .direct "c", x
  endm
```

The *.command* macro is similar to a *.sim* macro except that it associates a gpsim command with a particular instruction. This is useful for changing attribute values at different points of the program.

4.1.3 .assert macro

```
; Assertion
.assert macro x
 .direct "a", x
 endm
```

The *.assert* macro provides a source code mechanism for setting breakpoints (see chapter 5). An assertion is an expression associated with a specific instruction. It essential means, "If the expression at this instruction evaluates to false, then halt the simulation."

```
; Close the switch because of capacitance portc1 will go high after a delay:
   ; R=145, C=4.2e-6 TC=6.11e-4 or 1527 cycles 0-2 volts requires 0.51 Tc
   .command "SW1.state=true"
        nop
        ; portc0 should be same as portc1
   .assert "(portc & 3) == 0, \"SW1 closed, cap holds low\""
        nop
```

In this example, the *.command* macro writes to the switch module's *.state* attribute (see section 8.1.4). Just prior to executing the first nop instruction, the switch will be closed. The *.assert* macro at the very next instruction makes sure that the expected state is seen on PORTC.

4.2 Sockets

gpsim supports a socket interface. This is inhibited by default. Advanced users may wish to study code in the *examples/scripts* subdirectory. This code not distributed and is only available in the subversion repository.

Chapter 5

Assertions and Extended Breakpoints

gpsim supports a wide variety of breakpoints and assertions. Many of these were described with the break command. This section will illustrate how to extend the break command even further and introduce simulation assertions.

Breakpoint Messages

A breakpoint message is an ASCII string that is displayed whenever a breakpoint is encountered. Any break point can have an associated message. The syntax at the command line is

```
break conditions, "This is a breakpoint message"
```

The conditions are described above in the break command and are the conditions under which the break occurs.

Breakpoint messages are useful for distinguishing among many different breakpoints.

```
break w counter & 0xf0 == 0x80, "Counter overflowed!"
```

In this example, the user is monitoring the upper nibble of the variable counter and breaking whenever it is equal to 8. When the command is entered, gpsim will display:

```
break when bit pattern 1000XXXX is written to register counter(0x26). break #: 0x20
```

The breakpoint can be queried with the break command:

```
gpsim> break 32
32: p18f452 register write value: [0x26] & 0xf0 == 0x8
    Message:Counter overflowed!
```

When the simulation encounters the break, execution halts and the message is printed.

5.1 Assertions and Embedded Simulation commands

gpsim's breakpoint design is a powerful tool that can catch many problems. The assertion design extends this power even further. An assertion is like a breakpoint that is associated with a particular instruction. For example, you may have a routine that requires BANK 0 be selected. A gpsim assertion can be placed at the entry of the routine to verify that this is the case.

```
.assert  "(status & 0x60) == 0, \"Bank 0 must be selected!\""
```

The syntax is identical to the extended breakpoint command. The expression is the condition that is checked. If the expression evaluates to false, then the code halts and prints the message. The *.assert* is a macro that is part of gputils. It requires a string as its input argument. Notice that the assertion message is embedded in the argument. gpasm and MPASM copy C's method of placing a backslash in front of quotations that are part of a string.

Command Assertions

A command assertion is a gpsim associated with a particular instruction in your PIC source code. These are useful for changing the behavior of the simulation based on where the code executes. Almost any gpsim command can be placed in a command assertion. However, the most useful ones are assignment commands. For example:

```
.command "SW1.state = false"
```

This assignment writes to the state attribute of a switch module named SW1.

Chapter 6

Trace and Log: 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	adresses 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
0x00000000000026F6 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) 0x00000000000026FA p18f452 0x0014 0x0004 clrwdt
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.

Saving Trace to a file

The trace buffer may contain thousands of entries making it difficult to search. The trace save feature will allow the trace buffer to be written to a file.

```
gpsim> trace save mytrace.log
```

The entire contents of the trace buffer are decoded and written to the file. The format of the trace is the same as it is when displayed at the command line.

Raw Traces

The *raw* trace buffer is the trace buffer displayed in a minimally decoded form. This is primarily used for gpsim development. When saved to a file, the raw trace is not decoded at all. In addition, the processor's state is written to the file. Thus third party tools can be written to create custom trace reports[1].

[1] FIXME - The dynamically created trace type information needs to be written to this file too. Without it, it is difficult to tell what each traced item is.

Chapter 7

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 a PIC16C84 can only drive low. Multiple I/O pins may be 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.

7.1 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 nodes[1]. 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:

[1]Although, 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.

7.1.1 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 characterised by a voltage and an impedance. When a node is updated, gpsim performs a Thevenin voltage summing of all the stimuli together. The resultant voltage is then propagated to all connected stimuli as the current state of the node.

For example, in the port A open collector / port B weak pull-up connection example, gpsim assigns a voltage of 5V with an impedance of 20kohms to the pull up resistor, and a voltage of 0V with an impedance of 150ohms to the open collector if it is active, or 100Mohms if it's not driving. The Thevenin sum will be roughly 0.05V if the output is driving, or 5V otherwise. Capacitive effects are not currently supported.

7.2 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.

7.3 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
```

7.3.1 Analog Asynchronous Stimuli

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

7.4 Extended Stimuli

Discuss the extended stimuli in the modules/ directory. In particular, describe the *PulseGen* module and how it can complete replace the asynchronous stimuli. Also describe the *PullUp* and *PullDown* modules and how they can be manipulated into being general purpose DC voltage sources (FIXME, would it make sense to rename these modules?).

Chapter 8

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.

Modules reside in a library and are dynamically loaded with the *module* command. All modules have I/O pins which can connect to other modules or processors. Most modules provide *attributes* that allow the user to control a module's behavior or query its internal state. For example, the USART module has transmit and receive baud rate attributes that may be configured:

```
gpsim> U1.txbaud = 9600   # set the transmit rate
gpsim> U1.rxbaud          # query the receiver rate
9600
```

The symbol command can be used to query all attributes of a module.

```
gpsim> symbol U1.     # note the period
U1 = USARTModule
U1.console = false
U1.crlf = true
U1.loop = true
U1.rx = 0
U1.rxbaud = 9600
U1.tx = 0
U1.txbaud = 9600
U1.xpos = 72.00000000000000
U1.ypos = 276.0000000000000
```

Modules may provide only help which can be accessed using the help command:

```
gpsim> help U1
USARTModule
no description
```

Well, the USART module isn't the best example here! However, a better example is one of the USART attributes.

```
gpsim> help U1.txbaud
9600
USART Module Transmitter baud rate
```

8.1 gpsim Modules

gpsim provides a library of useful modules for simulation. The current version includes the following modules:

pushbutton	
pullup	A resistor connected (nominally) to Vdd
pulldown	A resistor connected (nominally) to Vss
usart	A serial interface with a GUI terminal window
pulsegen	
I2C eeprom	
switch	Switch, which connects two nodes together
and2	2-input logical AND gate
or2	2-input logical OR gate
xor2	2-input logical XOR gate
not	Inverter (logical NOT gate)
led_7segments	A 7-segment LED digit
led	
TTL377	A 74HC377 style 8-bit tristate latch
Encoder	

8.1.1 USART

The USART module is a full duplex configurable USART. In graphics mode, the USART will display its output in a console. In addition, the console will accept keyboard input.

Attributes

.tx - The *.tx* attribute is the USART transmit register. Data written to this attribute will initiate a transmission. The USART does not support a transmit FIFO.

.rx - The *.rx* attribute is the USART receiver register. Data received by the USART is available for querying through here.

.txbaud - The *.txbaud* attribute specifies the transmitter baud rate.

.rxbaud - The .rxbaud attribute specifies the receiver baud rate.

.console - When set to *true*, the console window will display received data and will accept keyboard entries for the transmitter.

.crlf - When set to *true*, carraige returns and line feeds generate new lines in the console window.

.loop - When set to true, received characters are looped back to the transmitter.

.xpos - horizontal position in breadboard window.

.ypos - vertical position in breadboard window.

I/O Pins

.TXPIN - transmit pin

.RXPIN - receiver pin

.CTS - Clear to send pin. This can be left unconnect

.RTS - Request to send pin.

8.1.2 Logic

The only attributes supported be the logic devices are the standard *.xpos* and *.ypos* breadboard positions. FIXME There should be attributes to specify the switching characteristics.

and2 - Two input AND gate

I/O pins

.in0 - First input

.in1 - First input

.out - Output

or2 - Two input OR gate

.in0 - First input

.in1 - First input

.out - Output

xor2 - Two input XOR gate

.in0 - First input

.in1 - First input

.out - Output

not - Inverter

.in0 - Input

.out - Output

8.1.3 I2C EEPROM

There are currently three I2C EEPROMs supported: I2C-EEPROM2k, I2C-EEPROM16k, and I2C-EEPROM256K. FIXME. Probably should have attributes for configuring the prom size. Also, there's currently no way to query or modify the prom contents.

I/O Pins

.A0 -

.A1 -

.SCL -

.SDA -

.WP

8.1.4 Switches & Resistors

The *switch* module is a model of a simple two terminal switch. It may be controlled either from the command line or the breadboard gui. The *switch* module's open and closed resistance are controlled by attributes. Thus a two terminal resistor can be modeled as a switch that is always closed (or open).

Attributes

.Ropen - Switch resistance in ohms when the switch is opened.

.Rclosed - Switch resistance in ohms when the switch is closed.

.state - Switch state. When *false* the switch is open. The *.state* attribute is writable.

I/O Pins

.A - One side

.B - The other side

8.1.5 Voltage Sources, Resistors, and Capacitors

The *pullup* and *pulldown* modules are two terminal devices with one terminal tied to a voltage source. Their voltage, resistance, and pin capacitance are controllable via attributes.

Attributes

.voltage - DC voltage

.resistance - resistance in ohms between the I/O pin and the voltage source.

.capacitance - capacitance in farads between the I/O pin and ground.

I/O Pins

.pin - the only pin exposed.

8.2 Third Party Modules

In addition to the standard modules, third are separately distribute gpsim modules.

8.2.1 Character LCD - HD44780

This module emulates the ubiquitous HD44780 character LCDs.

8.2.2 Graphic LCD - SED1530

This module emulates a 100X32 pixel graphics LCD based on dual SED1350 controllers.

8.3 Writing new modules

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

The easiest way to write a new module is to start from the source code from one of the existing modules. For example, suppose your project produces a serial bit-stream in PPM coding and you want to display the output during the simulation. The external module you need is similar to the usart module but not the same, so start by making a copy of the usart module and then modify it to work how you need.

To be able to load your module into gpsim it needs to be in a library. Usually you will be creating a new library just for one device, but sometimes you'll have a few devices.

Either way, the library must declare to gpsim what devices it contains. This is achieved with an array of Module_Types class instances, returned to gpsim by a function named “get_mod_list”. All gpsim module libraries must declare this function. You can copy the required template from the gpsim source – probably one of the “extras” modules is slightly cleaner than the main library. For our PPM decoder example, we might have a module_manager.cc containing the following code:

```
/* IN_MODULE should be defined for modules */
#define IN_MODULE
#include <stdio.h>
#include <gpsim/modules.h>
#include "ppm.h"
Module_Types available_modules[] =
{
  { "ppm_display", "ppm_rx_iface", PpmDisplay::construct},
  // No more modules
  { NULL,NULL,NULL}
};
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
/*****************************************************************
 * get_mod_list - Report all of the modules in this library.
 *
 * This is a required function for gpsim compliant libraries.
 */
  Module_Types * get_mod_list(void)
  {
    return available_modules;
  }
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

This declares that this library provides one module, called ppm_display, implemented by the C++ class PpmDisplay. The class which implements the module must provide a static method “construct” to create a new instance of the class. For example:

```
Module * PpmDisplay::construct(const char *_new_name=0)
{
    PpmDisplay *ppmd = new PpmDisplay(_new_name);
    ppmd->create_iopin_map();
    ppmd->create_window(_new_name);
    return ppmd;
}
```

Your module will need to include stimuli for its I/O connections. You can use the standard gpsim stimulus classes: IOPIN, io_bidirectional, io_bidirectional_pu, io_open_collector. In many cases, however, you will want to derive your own class from one of them. This will allow you to customise the actions when the node state changes. For example:

```
class DecoderPin : public IOPIN
{
private:
    PpmDisplay * Parent;
public:
    DecoderPin ( PpmDisplay * parent, unsigned int b, const char * name=0 );
    virtual void setDrivenState(bool new_state);
};
```

The only methods we provide here are the constructor and an overridden “setDrivenState”. This is because our PPM decoder needs to be told when the input pin changes state.

Chapter 9

Symbolic Debugging

gpsim maintains a symbol table.

<write me>

Chapter 10

Macros

<write me>

Chapter 11

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 12

The ICD- Not Supported in versions 0.21.0 and later

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 13

Examples

The *examples/* subdirectory contains several examples. The *examples/projects/* subdirectory demonstrate sample projects that can serve as templates for new projects. In addition, the *examples/modules* subdirectory contains several examples illustrating how to use gpsim's various modules. Finally, as described in chapter 14, gpsim's regression tests illustrate many powerful debugging techniques that have not been fully documented.

usart_gui example

Each example contains a brief *README* explaning its purpose. For example, the *README* for the *usart_gui* example in the *examples/modules* directory contains

```
The tests the USART module with the GUI fix.
The code for a 16f628 PIC is used. The code first transmits a string of
characters, which are instructions to the user, to the USART module which
will then be displayed on its GUI. This verifies that the USART can receive
serial data.
When the focus is on the USART GUI window, characters typed on the keyboard
are sent from the USART to the PIC and then retransmitted from the PIC back
to the USART.
If all works, the typed characters will be displayed in the GUI text window
of the USART. Both transmit and receive must be functioning for this to
happen.
```

Fixme - we really need to document all of the examples!

Chapter 14

Regression Tests

Starting with version 0.22.0, gpsim distributes regression tests. The purpose of a regression test is to validate correctness. The tests are designed to exercise many of the aspects of gpsim and gpsim's modules. While designed primarily for developers though, the regression tests also serve as a rich source of examples. There are many features gpsim's developers will tuck away into a regression test and fail to document!

Chapter 15

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.

15.1 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.

15.2 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.

15.3 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.)

15.4 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 eeadr 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
```

15.5 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 progam 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();

}
```

15.6 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.

15.7 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 breapoint 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, accept 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 function[1] 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.

[1] A 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

Index

and2, 45
attach, 9

break, 9, 10
bus, 9

Capacitors, 47
clear, 9, 12

disassemble, 9, 12
dump, 9, 12

echo, 12

frequency, 9, 13

GNU, 62

HD44780, 47
help, 9, 13

I2C EEPROM, 46
icd, 9, 13
instructions, 57

License, 62
list, 9, 13
load, 9, 13
log, 9
Logic, 45

macros, 14
module, 9, 15
Modules, 43

NO WARRANTY, 62
node, 9, 17
not, 46

or2, 45

processor, 9, 17

quit, 9, 17

registers, 58
Resistors, 46, 47
run, 9, 18

SED1530, 47
set, 9
step, 9, 18
Stimulus, 39
stimulus, 9, 18
stopwatch, 9, 19
Switches, 46
symbol, 9, 18

trace, 9, 20

USART module, 44

version, 9
Voltage Sources, 47

x, 9, 20
xor2, 46