gpsim

Т. Скотт Даттало

05 ИЮНЯ 2005

Содержание

Введение

gpsim – это программный симулятор, с полной поддержкой всех свойств PIC микроконтроллеров Microchip, распространяемый под GNU General Pablic License (смотрите раздел COPYING).

gpsim был разработан со всей возможной точностью. Точностью, включающей весь PIC – от ядра до выводов ввода-вывода, включая ВСЮ внутреннюю периферию. Таким образом, возможно открыть стимулы и связать их с выводами ввода-вывода, и проверить PIC, тот самый PIC, и тем же манером, как вы сделали бы в реальном мире.

gpsim был разработан, чтобы быть столь быстрым, сколь возможно. Работает симуляция реального времени со скоростью в 20 МГц.

gpsim может управляться либо через графический пользовательский интерфейс (GUI), через интерфейс командной строки (CLI), либо с помощью процесса управления. Типичные средства отладки, подобно точкам останова, пошаговому режиму, дизассемблированию, проверке памяти и модификации, и т. д., поддерживаются все. Вдобавок, такие комплексные отладочные средства, как трассировка реального времени, заявки, условные остановки и подключаемые модули для нескольких наименований, также поддерживаются.

Часть 1

gpsim – Обзор

Если вы не намерены брести через все детали, то этот раздел поможет вам получить необходимые знания для начала работы. Файлы INSTALL и README предоставят больше последней информации, чем этот документ, поэтому, пожалуйста, обратитесь сначала к ним.

1.1 Создание исполняемого фай ла

Исполняемый файл gpsim создается на манер других программ, представленных исходными текстами:

Команда	***Описание***
tar -xvzf gpsim-x.y.z.tar.gz	разархивирует сжатый tar файл
./configure	создает “makefile” уникальный для вашей системы
make	компилирует gpsim
make install	устанавливает gpsim

Последний шаг требует привилегий root.

1.1.1 Задание деталей - ./configure опции

gui-less

Конфигурация по умолчанию будет поддерживать gui (графический пользовательский интерфейс). Но также доступен cli (интерфейс командной строки), поскольку есть много людей, предпочитающих пользоваться им. Эти стойкие души могут построить интерфейс только командной строки, конфигурируя gpsim:

./configure –disable-gui

debugging

Если вам угодно отладить gpsim, тогда вы, возможно, воспользуетесь gdb. Следовательно, вы захотите отвергнуть общие библиотеки:

./configure –disable-shared

Этим создается единственный, но большой исполняемый файл с символьной информацией.

1.1.2 RPMs

gpsim также распространяется в форме RPM. В текущих версиях два RPM – gpsim-devel и gpsim. Оба должны быть установлены. Есть так же RPM для исходных кодов. Он может использоваться для построения двоичного RPM уникального для вашей системы. Обратите, пожалуйста, внимание на последние INSTALL и README для получения новейшей информации.

1.1.3 Windows

gpsim работает и на Windows. Борут Разем создал gpsim Windows web-сайт:

http://gpsim.sourveforge.net/gpsimWin32/ gpsimWin32.html

Вы можете найти детальные инструкции по установке gpsim и ее зависимостям. Внешний вид можно найти:

http://gpsim.sourveforge.net/snap.php

1.2 Запуск

Исполняемый файл, открытый как описано выше, называется: gpsim. Следующие опции командной строки могут быть установлены, когда gpsim запускается.

```
gpsim [-?] [-p <device> [<hex_file>]] [-c <stc_file>]
  -p, --processor=<processor name>     processor (иначе -pp16c84 для 'c84')
  -c, --command=STRING                 startup командный файл
  -s, --symbol=STRING                  .cod символьный файл
  -L, --sourcepath=STRING    через двоеточие список директорий поиска
```

```
 -v, --version              gpsim версия
 -E, --echo                 строка эхо командной линии в консоль
 -i, --cli                  режим только командной линии
 -S, --source=STRING     'enable' или 'disable' загрузки исходного кода.
                            По умолчанию 'enable'.
                            Полезно для запуска быстрых регрессионных
                            тестов.
 -d, --icd=STRING           использовать ICD (иначе, -d /dev/ttyS0).
 -D, --define=STRING     определить символ со значением, добавленным
                            в gpsim таблицу символов. Определяется любое
                            количество символов.
 -e, --exit=STRING          Приводит к авто выходу gpsim
                            по условию. Задание onbreak приведет
                            к выходу gpsim, когда симуляция
                            останавливается, но не раньше, чем текущий
                            командный скрипт завершится.

Examples:
  gpsim -s myprog.cod          <-- loads a symbol file
  gpsim -p p16f877 myprog.hex  <-- select processor and load hex
  gpsim -c myscript.stc        <-- loads a script

Help options:
  -?, --help                   Показывает это сообщение help
  --usage                      Отображает краткую подсказку
```

Обычно gpsim запускается следующим образом:

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

([My-Computer]$ текст – пример типичной командной строки bash – вы только вводите текст после этого приглашения). Команда загружает .cod файл (mypic-program.cod), созданный с помощью gputils. Под Windows gpsim также может открыть окно DOS или CygWin сессию bash (терминальный интерпретатор командной строки), и запустить gpsim из него.

1.3 Требования

Программа gpsim была разработана под Linux. Она должна встраиваться и запускаться прекрасным образом под всеми популярными дистрибутивами Linux, подобными Redhat. gpsim также был портирован в MAC, MicroSoft Windows, Solaris и BSD. Два пакета gpsim требуются постольку, поскольку не все дистрибутивы Linux имеют readline и qtk (инструментальные средства gimp). Скрипт ./configure подскажет, если эти пакеты не установлены на вашей системе, или устарели.

Необходимо удовлетворить минимальные требования по оборудованию для запуска gpsim. Но чем оно быстрее, тем лучше!

Утилиты gputils и gnupic также очень полезны. gpsim может использовать сразу hex-файлы, но, если вы хотите использовать символьную отладку, тогда вам потребуется .cod-файл, который производит пакет gputils. cod-файл имеет тот же формат, что производится MPASM.
(.cod-файл – это символьный файл созданный ByteCraft и используемый Microchip, а MPASM – ассемблер Microchip).

Часть 2

Интерфейс к омандной строк и

Интерфейс командной строки в известной мере первичен. Таблица ниже суммирует допускаемые команды. Краткое описание этих команд может быть выведено при вводе help в командной строке.

Команда	***Суммарно***
attach	присоединяет стимул к узлу
break	устанавливает точку останова
bus	добавляет или показывает базовый узел
clear	удаляет точку останова
disassemble	дисассемблирует текущий cpu
dump	отображает RAM или EEPROM
frequency	устанавливает частоту процессора
help	введите help для подсказки
icd	поддержка отладки в схеме
list	отображает исходные и list файлы
load	загружает либо hex, либо командный файл
log	Log/record события в файл
node	добавляет или отображает узлы стимула
module	Выбор&Отображение модулей
processor	Добавить/Список процессоров
quit	покинуть gpsim
reset	сбросить все или часть симуляций
run	выполнить pic программу
set	отображение и управление флагами gpsim
step	выполнить одну или более инструкций
stimulus	открыть стимул
stopwatch	измерение времени между событиями

Команда	*Суммарно*
symbol	Добавить/Список символов
trace	дамп последовательности трассировки
version	отображает версию gpsim
x	проверка и/или модификация памяти

Встроенный help предоставляет дополнительную online информацию.

2.1 attach

attach node1 stimulus1 [stimulus2 stimulus_N]

attach используется для определения связи между между стимулами и узлами. Хотя бы один узел и один стимул должны быть определены. Если определено больше стимулов, тогда они все будут связаны с узлом, например:

```
gpsim> node n1                    # Определяет новый узел
gpsim> attach n1 porta4 portb0    # Связывает два вывода I/O с узлом
gpsim> node                       # Отображает новый “список
связей”
```

2.2 break

Команда break используется для установки и проверки точек останова. Новым точкам останова присваиваются уникальные номера. Эти номера могут быть использованы при запросах или удалении точек останова. Точки останова останавливают симуляцию, когда ассоциированные с ними условия выполняются (становятся истинными). Точки останова игнорируются в пошаговом режиме.

Проверка точек останова

break [bp_number] (номер точки останова)

Точки останова могут проверяться командой break без дополнительных опций. Определенная точка останова может быть проверена заданием номера.

Память программы/Выполнения остановки (Program Memory/Execution breaks)

Более общие точки остановки – это точки остановки выполнения. Последние останавливают выполнение в тот момент, когда программный счетчик достигает адреса, на котором установлена точка останова. Синтаксис следующий:

break e | r | w ADDRES [expr (выражение)]

Симуляция останавливается, когда адрес выполнен, прочитан или записан. ADRESS может быть символьным или числовым. Если дополнительное выражение определено, тогда оно должно стать истинным, до остановки симуляции. Опции чтения и записи относятся только к тем процессорам, которые могут работать с их собственной программной памятью.

Точки останова регистровой области памяти (Register Memory)

gpsim может также ассоциировать точки останова с доступом к регистрам. Это полезно для выявления ошибок, которые связаны с RAM. То есть, вы можете сказать что-нибудь вроде “остановить выполнение, когда бит 4й регистра 42 сбрасывается”. Синтаксис команды:

break r | w REGISTER [expr]

Симуляция останавливается, когда REGISTER читается или записывается, и дополнительное выражение становится истинным. Поддерживаются два стиля выражений. Один обращается только к выражениям REGISTER, другой проверяется комплексно. Пример ниже иллюстрирует разницу. Вот пример останова при записи в регистр. Он остановит симуляцию, если

любое значение записывается в регистр для переменной, названной *temp1*.

```
break w temp1
```

Вот условная запись для случая только некоторых значений:

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

Вот условие, затрагивающее определенные биты:

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

Остановка происходит только в случае, когда шестнадцатеричное число "C" записывается в старшие биты переменной *temp1.*

Булевы выражения

Иногда необходимо определить вспомогательное условие с точкой останова. Например, есть временная переменная, которая обобщается через код. Вы можете захотеть прервать запись в эту переменную только, когда выполняется определенная подпрограмма. Например, следующая точка останова переключается, когда *temp1* записывается, и пока счетчик программы находится между метками *func_start u func_end:*

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

ПРЕДУПРЕЖДЕНИЕ: Используйте этот тип точки останова, если вы подозреваете прерывание программы через запись в переменную.
Другой ситуацией может оказаться такая, где вы хотите прервать запись в переменную только, если какая-то другая переменная имеет то же значение:

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

Если программа записывает в переменную *temp1*, тогда симуляция останавливается, если младшие биты CurTask не эквивалентны 5.

Точки останова атрибута

gpsim также поддерживает концепцию точек останова атрибута. Атрибуты – параметры, которые gpsim и ее модули предоставляют пользовательскому интерфейсу. Например, остановка наблюдения симулятора предоставляет атрибуты, которые поддерживают точки останова. Это свойство предназначено в основном для создателей модулей, для поддержки механизма, который позволяет пользователю управлять модулем.

2.3 clear

clear bp_number

Команда *clear* используется для удаления точек останова. Номер точки останова должен быть задан. Команда *break* без аргументов отображает все в настоящий момент определенные точки останова. Это может использоваться для уточнения номера точки останова. При удалении, точка останова удаляется.

2.4 disassemble

disassemble [begin:end] | [lenght]]

Команда *disassemble* раскодирует *opcodes* программной памяти в их стандартную мнемонику. Без опций команда дизассемблирует инструкции вокруг текущего значения счетчика программы:

```
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)
```

С единственной числовой опцией команда дизассемблирования будет

2.5 dump

dump [r | e]

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

Обратитесь к команде 'x' для проверки и модификации конкретных регистров.

2.6 echo

Команда echo используется подобно заявлению печати в файлах конфигурации. Она только позволяет вам отобразить информацию о вашем файле конфигурации.

2.7 frequency

Эта команда устанавливает тактовую частоту. По умолчанию gpsim использует частоту 4 МГц. Частота используется для вычисления времени в секундах. Используйте эту команду для задания значения. Если не предоставлено никакого значения эта команда печатает текущее время. Заметьте, что PIC имеют инструкцию clock, которая продвигает внешнее время. Это значение внешнего времени.

2.8 help

Само по себе help будет отображать все команды вместе с краткими описаниями, как они работают. Команда 'help' предоставляет более богатую online помощь. Команда может также отображать информацию об атрибутах.

2.9 icd

icd [open <port>]

Команда open используется для включения режима ICD и определяет последовательный порт, в который включено ICD (то есть, “icd open /dev/ttyS0”). Без опций (и после включения icd) она будет печатать некоторую информацию об ICD.

2.10 list

list [[s | l] [*pc][line_number1 [,line_number2]]]

Отображает содержимое исходного файла и файла листинга. Без опций команда будет использовать последние определенные опции.
list s будет отображать строки в исходном (или .asm) файле.
list l будет отображать линии в .lst файле.
list *pc будет отображать либо .asm, либо .lst линии вокруг pc (счетчика программы).

Команда *list* позволяет вам видеть исходный код при отладке.

2.11 load

Команда *load* используется для загрузки hex-файла, файла конфигурации или .cod файла. Hex-файл обычно используется для программирования физической части. Следовательно он не поддерживает символьную информацию. .cod файл, с другой стороны, поддерживает символьную информацию. Единственная причина для использования hex-файла, полное отсутствие .cod файла.
Синтаксис загрузки файла исходного кода:

load [processortype] file

gpsim автоматически определит, файл hex или .cod формата. Опция processortype позволит переопределить процессор, обозначенный в .cod файле.

Файл конфигурации – это скрипт, содержащий gpsim команды. Крайне полезно создавать окружение отладки, которое будет использовано повторно.

2.12 macros

Макросы определяются подобно:

```
name macro [arg1, arg2, ..., argN]
        macro body (тело макроса)
endm
```

И они вызываются:

```
name param1,  param2, ...,  paramN
```

Макросы – способ коллекционировать некоторые параметризованные команды в одной короткой команде. Первая строка определения макроса определяет имя макроса и дополнительные аргументы. *name* используется для вызова макроса. Аргументы – текстовая строка удерживающая место. Когда макрос вызывается, параметры соотносятся с аргументами. То есть, *param1* в вызове может быть ассоциирован с *arg1* в определении. Параметры замещаются аргументами в теле макроса.
В следующем примере переменная или атрибут названный *mac_flags* будет использован в выражении. Аргументы *add* и *mask* появляются в теле макроса и предоставляют параметризованный путь для изменения этого выражения.

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

Заметьте, что идентификация произвольна. Макрос вызывается:

```
mac_exp 1, 0b00001111 # наращивает младшие биты
```

Параметр *add* заменен числом 1, тогда как *mask* замещена бинарным

числом *0b00001111*. Вызов возвращает в gpsim команду:

```
mac_flags = (mac_flags+1) & 0b00001111
```

Вложенные макросы

Тело макроса может состоять из любых команд gpsim. Частный интерес представляет вызов макроса внутри другого макроса. Вот другой макрос, который вызывает макрос, определенный выше:

```
# Пример вложенного макроса
macl macro p1, p2
        run
        mac_exp p1, p2
endm
```

И он может использоваться подобным образом:

```
macl 1,         0b00001111 # проверяет младшие биты
macl (1<<4), 0b00001111 # проверяет старшие биты
```

Первый вызов начинает симуляцию с выполнения команды *run*. Когда точка останова обнаруживается, управление возвращается командной строке, и вызывается макрос *mac_exp*.

Отображение определенных макросов

Все определенные к настоящему времени макросы могут отображаться вводом команды *macro* без имени или аргументов:

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

2.13 module

Команда *module* используется для загрузки и запросов к внешним модулям. Модуль – специальная часть программы, которая может расширять gpsim некоторым образом. Индикаторы и переключатели – примеры модулей. Библиотека модулей – коллекция модулей.

Загрузка библиотек модулей

```
module lib lib_name
```

Опция *lib* используется для загрузки библиотеки модуля. Библиотеки модуля – зависимые от системы общие библиотеки. На Windows – это DLL, а в UNIX – общие библиотеки. Это означает, что либо библиотеки должны располагаться в директории, где ОС знает, что там расположены библиотеки, или следует задать полный путь для *lib_name*. gpsim предоставляет библиотеку модулей с несколькими модулями:

```
gpsim> module lib libgpsim_modules
```

Отображение доступных модулей

```
module list
```

Опция *list* отобразит все модули, которые могут быть загружены. Вот пример встроенных gpsim модулей.

```
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

Загрузка обозначенного модуля

module load module_type [module_name]

После того, как библиотека была загружена, обозначенный модуль может быть проиллюстрирован. module_type – это то, что отображено командой module list. В качестве опции может выступать имя модуля. Вот пример:

gpsim> module load led D1

Отображение загруженных модулей
Запрос модулей

2.14 node

node [new_node1 new_node2 ...]

Если не обозначено new_node, то все узлы, которые были определены, отобразятся. Если new_node обозначено, тогда он будет добавлен в список узлов. Посмотрите на команды 'attach' и 'stimulus', чтобы увидеть, как стимул добавляет узлы.

Примеры:

node // отображает список узлов
node n1 n2 n3 // открывает и добавляет 3 новых узла в список

2.15 processor

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

Команда *processor* используется либо для определения нового процессора, либо для запроса одного из уже определенных. Обычно нет необходимости специально определять процессор, поскольку символьный файл уже содержит эту информацию. Но есть два исключения – символьная информация не доступна, или вы хотите переопределить процессор, обозначенный в файле. (Посмотрите команду load для понимания, как процессор может быть переопределен).
Чтобы посмотреть список процессоров, поддерживаемых gpsim, введите *'processor list'*. Для отображения состояния вводов-выводов процессора – '*processor pins'*. Вот, например, следующее отобразит номера выводов и их текущее состояние.

Примеры:

```
processor                   // Отображает уже определенные процессоры
processor list              // Отображает список поддерживаемых
processor pins          // Отображает цоколевку и состояние выводов
processor p16cr84 fred      // Создает новый процессор
processor p16c74 wilma      // и другой
processor p16c65            // Создает процессор без имени
```

2.16 quit

Покидаем gpsim.

2.17 run

Начинает (или продолжает) симуляцию. Симуляция будет продолжена пока следующая точка останова не обнаружится.

2.18 step

step [over | n]

нет аргументов: шаг на одну инструкцию
числовой аргумент: шаг на количество (число) инструкций
аргумент 'over': шаг через следующую инструкцию

2.19 symbol

symbol [symbol_name[symbol_type value]]

Команда symbol используется для запроса и определения символов. Если нет опций, отобразится вся таблица символов. Создание определенных пользователем символов ограничено в настоящий момент (сверьтесь с 'help' для уточнения положения дел).

2.20 stimulus

stimulus [{type} options]

Команда stimulus создает сигнал, который может быть привязан к узлу или атрибуту. Если нет опций, тогда отображаются все в настоящий момент определенные стимулы.
Заметьте, что в большинстве случаев легче создать файл стимула, чтобы не вводить команду вручную.

состояние_при_инициализации	*состояние при старте и выполнении*
start_cycle	cycle симуляции, когда начинается стимул
period	период стимула
name	определенное имя стимула

Вот пример стимула, который будет генерировать два импульса и повторение этого через 1000 циклов.

```
stimulus asynchronous_stimulus # Состояние инициализации AND
                               # состояние стимула при выполнении
initial_state 0
start_cycle 0 # асинхронный стимул будет выполнен в 'period' циклов.
              # Удалите эту линию, если не хотите выполнения.
```

```
period 1000
{   100,  1,
    200,  0.
    300,  1,
    400,  0
} # Даем имя стимулу: name two_pulse_repeat
end
```

Стимул может быть запрошен введением его имени в командной строке:

```
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
statetes = 5
          100  1
          200  0
          300  1
          400  0
          1000  0
initial=0
period=1000
start_cycle=0
Next break cycle=100
```

Даже в этом примере, использующем 1 и 0 для данных, можно вместо них применить целые, с плавающей точкой числа, или выражения. Целые полезны для привязки стимулов к атрибутам. Выражения полезны для абстрагирования данных. Загляните в Часть 6, где больше внимания уделяется обсуждению, и приведено больше примеров стимулов.

2.21 stopwatch

2.22 trace

```
trace [dump_amount]
```

trace будет распечатывать самые последние “dump_amount” трассировки.

Если нет специфицированных dump_amount, тогда будет отображен полный буфер трассировщика.

2.23 version

version

Отображает версию gpsim. Заметьте, что эта команда будет, возможно, получена через атрибут с тем же (или подобным) именем.

2.24 x

x [file_register][new_value]

опции:

file_register – место ram, которое будет проверено или модифицировано
new-value – новое значение, записанное в file_register.
Если опции не заданы, все содержимое file_register'ов будет отображено (dump).

Часть 3

Графическ ий поль зователь ский интерфейс

gpsim предоставляет также графический пользовательский интерфейс, который несколько облегчает жизнь по сравнению с cli. Возможно открыть окно для обзора всех деталей вашего окружения отладки. Чтобы получить максимум от вашей сессии отладки вам следует откомпилировать ваш код с помощью gpasm (gnupic ассемблер) и использовать символьные .cod файлы, им производимые.

3.1 Главное окно

3.1.1 Меню

File->Open	.stc или .cod файлы
File->Quit	покидаем gpsim
Windows->*	Окрываем/Закрываем окна

3.1.2 Клавиши

(Так же есть привязка к клавиатуре в исходном окне)

Step	Шаг на одну инструкцию
Over	Шаг до перехода pc к следующей инструкции
Finish	Выполнение до возврата к адресу
Run	Постоянное выполнение
Stop	Остановка выполнения
Reset	Сброс CPU

3.1.3 **Режим симуляции**

Этим управляется то, как работает gpsim, и как обновляется GUI.

Never	Не обновлять GUI при симуляции. Это самый быстрый режим. Вы должны остановить симуляцию нажатием Ctrl-C в интерфейсе командной строки.
x cycles	Обновляет GUI каждые x циклов симуляции.
every cycle	Обновляет GUI в каждом цикле. (вы увидите все, если пополните запасы кофе :-)
x ms animate	Здесь вы можете замедлить симуляцию введением задержек между каждым циклом
realtime	Это даст возможность gpsim попытаться синхронизировать скорость симуляции с настенными часами

3.2 **Проводники исходного кода**

3.2.1 **.asm Проводник**

Когда загружен .cod файл с исходным кодом, что-нибудь появится в этом окне. Есть пространство слева от текста, где символы представляют программный счетчик (pc), точки останова, и т. д. Двойной щелчок в этом пространстве переключает точки останова. Вы можете перетаскивать эти символы выше или ниже, чтобы переместить их и изменить PC и переместить точку останова. Щелчок правой клавишей на тексте вызывает выпадающее меню с шестью пунктами (слово “здесь” в некоторых пунктах меню означает линию в тексте, на которую был установлен маркер мышки, когда правая клавиша была нажата):

Пункты меню	**Описание**
Find PC	Этот пункт будет искать PC и менять таблицу страниц и прокручивать обзор текста до текущего (значения) PC
Run here	Этим установится точка останова “здесь”, и стартует симуляция, пока не попадет на точку останова
Move PC here	Этим просто изменяется PC на адрес, который имеет “эта” линия текста.
Breajpoint here	Устанавливает точку останова “здесь”.
Profile start here	Устанавливает старт маркера для профилирования

	правила здесь.
Profile stop here	Устанавливает стоп маркера. (Посмотрите раздел по окну профилирования)
Select symbol	Этот пункт меню доступен только, когда некоторый текст выбран в текстовом поле. Что он делает, так это ищет в списке символов выбранное слово, и если находит, выбирает его в символьном окне. В зависимости от типа символа еще кое-что делается, тоже самое, что при выборе символа в символьном окне: Если это адрес, тогда opcode и проводник отображают этот адрес. Если это регистр, проводник регистров отмечает ячейку. Если это константа, адрес, регистр или порт, он отмечается в символьном окне.
Find text	Этим открывается диалог поиска. С каждым нажатием клавиши “Find” находится текущая страница блокнота, и текст на этой странице используется.
Settings	Диалог, в котором вы можете изменить шрифт.
Controls	Подменю, содержащее команды симуляции (также привязка клавиатуры (рекомендуется), или в основном окне).

Вот привязка к клавиатуре:

Клавиша	**Команда**
s,S,F7	Пошаговое выполнение
o,O,F8	Перешагнуть инструкцию
r,R,F9	Постоянное выполнение
Escape	Остановить симуляцию
f,F	Выполнять до возвращения адреса

3.2.2 **Проводник Opcode - .obj проводник**

Это окно имеет две таблицы. Одна с ячейкой памяти, адресом, шестнадцатеричным значением и декодированной инструкцией на линии, другая с программной памятью, отображенной по шестнадцать ячеек в строке

и конфигурируемым ASCII столбцом.
С таблицей ассемблера вы можете:

Переключать точки останова двойным щелчком по строке.
Использовать те же клавишные команды, что и в проводнике исходного кода.
Правым щелчком мышки получить меню, где можно изменить шрифт.

Таблица Opcode.

Здесь программная память организована в колонки по шестнадцать ячеек в колонке, а строк столько, чтобы поместить всю память.
Семнадцатая колонка отведена под ASCII представление памяти. Вы можете конфигурировать эту колонку для использования в трех разных режимах:

Один байт на ячейку
Два байта на ячейку, старшие биты впереди
Два байта на ячейку, младшие биты впереди

Вы можете изменить шрифт, используя меню “Settings” (Установки).
Вы можете задать точки останова на одном или больше (помечая мышкой выбранные ячейки) адресах меню, вызываемом кликом правой клавиши мышки.

3.3 Обозреватель регистров

В вашем распоряжении два похожих окна регистров. Один для RAM, другой для EEPROM данных, когда они доступны.
Здесь вы увидите все регистры текущего процессора. Щелчок на ячейке отображает его имя и значение над всей таблицей. Вы можете изменить значение здесь, или ввести его в ячейку таблицы.
Следующие процедуры могут выполняться с одним регистром или с несколькими. (Выбор нескольких регистров выполняется с удержанием левой клавиши мышки и перемещением маркера до нужного места, где клавиша отпускается).

Установить и сбросить точки останова. Используйте выпадающее меню, вызываемое правой клавишей мышки, где вы можете прочитать, записать, прочитать значение и записать значение точек

останова. Вы можете также “сбросить точки останова”, но, заметьте, что при “сбросе точек останова” каждая точка останова на регистрах будет удалена.
Установить и сбросить логи регистров. Вы можете прочитать лог, записать, прочитать/записать специфические значения и для выбранных бит с помощью специальной маски. Вы можете выбрать разные имена файлов через “set log filename...” (установить имя лог-файла). По умолчанию это “gpsim.log”. Вы можете выбрать LXT или ASCII формат. LXT может быть прочитан программой gtkwave. По умолчанию установлен ASCII формат.
Можно копировать ячейки. Копирование доступно перетаскиванием за рамку выбранной ячейки (ячеек).
Можно заполнить ячейки. Переместите мышь к нижнему правому углу окна выбранной ячейки, и перетащите ее. Содержимое ячейки будет скопировано в другие ячейки.
Можно наблюдать за ними. Выберите пункт меню “Add Watch” (Добавить наблюдение).

Ячейки имеют разный фон, в зависимости от того, что в них представлено:

Файл регистр (то есть, RAM): светло голубой
Регистры специальных функций (то есть, STATUS, TMR0): темно голубой
Переобозначенный регистр (то есть, INDF, расположенный по адресу 0x80, тот же, что 0x00): серый
Неправильный регистр: черный. Если все шестнадцать регистров в строке неисправны, тогда строка не показывается.
Регистр с одной или больше точкой останова: красный. Регистры с лог-файлами тоже красные.

gpsim динамически обновляет регистры в процессе симуляции. Регистры, которые меняют значение между обновлениями окна в процессе симуляции подсвечиваются синим цветом фона.
В меню есть пункт 'settings ', с помощью которого вы можете изменить шрифт.

3.4 Символьный обозреватель

В этом окне, как это следует из названия, отображаются символы. Все

регистры специальных функций допускают ввод в символьном проводнике.
Если вы используете .cod файлы, тогда вы дополнительно имеете файловые регистры (определенные в cblocks), эквиваленты и метки адресов.
Вы можете фильтровать некоторые типы символов, используя клавиши в верхней части окна, и вы можете сортировать строки кликом на клавишах колонок (читающие символы (symbol), тип (type), адрес (address).
Вы можете добавить символ в окно наблюдения щелчком правой клавиши мышки и выбором пункта “Добавить в окно наблюдения” (Add to watch window). Этим добавится RAM регистр с адресом, эквивалентным значению символа.
Символьный проводник связан с другими окнами. Например, если вы щелкните по символу и:

Если это адрес, тогда проводники opcode и исходного текста отобразят адрес.
Если это регистр, проводник регистров выберет ячейку.

3.5 Обозреватель наблюдения

Это не только окно вывода, что предполагает название. Вы можете и видеть, и менять данные. Двойной щелчок по биту меняет его значение. Вы добавляете переменные сюда, отмечая в проводнике регистров и выбирая “Добавить в наблюдение” (Add watch) из меню. Выпадающее меню по щелчку правой клавиши мышки имеет следующие пункты:

Удалить из наблюдения
Установить значение регистра
Сбросить точки останова
Установить остановку по чтению
Установить остановку по записи
Установить остановку по чтению значения
Установить остановку по записи значения
Колонки

Последнее открывает окно, где вы можете выбрать, какие из следующих данных отображать:

Точка останова

Тип
Имя
Адрес
Десятичное
Шестнадцатеричное
Bx (биты слова)

Вы можете сортировать список наблюдаемых щелчком по клавише колонки. Двойной щелчок сортирует список в обратном порядке.

3.6 Обозреватель стека

Это окно отображает текущий стек. Выбор ввода создает окно кода, отображающее адрес возврата. Двойной щелчок устанавливает точку останова на адрес возврата.

3.7 Макет (breadboard)

Здесь вы можете создавать/модифицировать и проверять окружение PIC. Выводы отображены, как стрелки. Направление стрелки показывает, это вывод ввода или вывода. Цвет стрелки отображает состояние (зеленый – низкое, красный – высокое).
Вы не можете иллюстрировать поведение процессора отсюда, вы должны сделать это из командной строки, или с помощью .stc файла.
Вы может создать узел щелчком по клавише “новый узел” (new node). (Узел – часть схемы, к которой вы можете присоединить стимулы). Вы можете видеть список созданных узлов в пункте “узлы” в верхнем левом окне дерева макета. Вы можете создать соединение с узлом, щелкнув по выводу, а, затем, щелкнув по клавише “Присоединить стимул к узлу”. Это выведет список узлов. Выберите один двойным щелчком по нему.
Если вы щелкните по выводу, уже присоединенному к узлу, вы увидите узел и его соединение в нижней левой части окна. Вы можете отключить стимул, щелкнув по нему, и нажав клавишу “удалить стимул” (remove stimulus).
Когда вам захочется добавить модуль к симуляции, вы вначале должны определить библиотеку, которая содержит требуемый модуль. Щелкните по “Добавить модуль” (add module) и введите имя библиотеки (то есть, “libgpsim_mofules.so”). Теперь вы можете щелкнуть по клавише “Добавить

модуль". Выберите нужный модуль из списка двойным щелчком по нему. Введите имя для модуля (оно должно быть уникально и не использоваться ранее). Теперь вы можете позиционировать модуль. Перемещайте указатель мышки туда, куда хотите добавить модуль и щелкните левой клавишей мышки. Если щелкнуть средней клавишей по выводу, вы увидите, как вывод присоединен. Нажмите "Трассировать все" (trace all) для того, чтобы увидеть все сразу, и "Очистить трассировку" для удаления всего (вы удалите только графическое представление, а не соединения!). Если трассировка не работает, постарайтесь разместить элементы так, чтобы оставалось побольше места вокруг выводов.
Когда вы все сделаете, вы можете сохранить все щелчком по клавише "Сохранить конфигурацию" (save configuration). Позже вы можете загрузить этот файл из командной строки подобно тому, как загружали .cod файл, и файл "mynet.stc":

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

Вы можете загрузить только .stc файл поскольку он не содержит тип процессора и код. Вы можете создать (в редакторе) ваш собственный .stc файл и в этом файле дать команду "load c mynet.stc" после загрузки .cod файла. Позже вам понадобится только загрузить этот файл (gpsim -c my_project.stc).

3.8 Обозреватель трассировки

Это окно показывает трассировку выполнения инструкций. См. 5.

3.9 Обозреватель профиля

В этом окне показывается счетчик для адресов программной памяти. Окно профиля должно быть открыто перед стартом симуляции, поскольку трассировка не установлена по умолчанию.

Профиль инструкций
Здесь показано значение времени выполнения каждой инструкции.

Профиль ряда инструкций
Здесь вы можете группировать ряд инструкций в один ввод.
Выпадающее меню (правый щелчок мышки) содержит:

Remove range Удалить ряд

Add Range ...	Открывает диалог, в котором вы можете добавить ряд
Add all lables	Добавляет все метки кода, как ряд
Snaphot to plot	Открывает окно, содержащее граф данных. Из этого нового окна вы можете также сохранить (postscript) или распечатать.

Профиль регистра

Здесь показано число чтений или записи сделанных симулятором в регистр.

Профиль программного блока

Здесь вы можете увидеть статистику времени выполнения для выбранного программного блока. Вы отмечаете точки входа и выхода в проводнике исходного текста (profile start/stop). Если блок, который вы хотите измерить, имеет несколько точек входа и/или точек выхода, тогда вы должны отмаркировать каждую точку входа, так же, как (и обязательно) каждую точку выхода. Иначе вы получите неверные данные.

Когда вы сделаете это, gpsim будет (с началом симуляции) сохранять времена выполнения этих блоков и вычислять мин/макс/среднее/и т. д. Вы можете также использовать пункт меню “Печатать распределение” (plot distribution) для открытия окна, показывающего гистограмму данных. Из этого нового окна вы можете также сохранить (в postscript) или распечатать ее.

Вы можете измерить период вызова переключением между точкой входа и выхода. Если также захотите измерить время от сброса (или эквивалентной точки) до первого входа, тогда вы должны здесь отметить точку входа.

3.10 Остановка наблюдения (stopwatch)

Это окно показывает счетчик циклов и переустанавливаемый счетчик. Счетчик циклов тот же, что и в окне регистров. Он, в основном, считает инструкции. Другой счетчик считает так же, как счетчик циклов, но может быть очищен щелчком по клавише “clear” (или преустановлен вводом числа в окне ввода). Индикатор вверх/вниз отмечает направление счета.

Прокручиваемое значение определяет пределы счетчика циклов (модульный счетчик). Например, если прокручиваемое значение должно быть 0x42, тогда в какой бы момент переустанавливаемый счетчик не достиг величины 0x42, он провернется к нулю. Если счетчик считает вниз, тогда после установки нуля следующее значение будет 0x41. Если вы не хотите, чтобы так было, установите значение прокрутки несколько больше.

Часть 4
Контроль за выполнением : Точки останова

Одним из сильных свойств gpsim является гибкость, предоставляемая точками останова. Большинство симуляторов ограничены в выполнении типов точек останова.
Если вы хотите установить точки останова на регистры, на выполняемые циклы, неправильное размещение программы, переполнение стека, и т. д., тогда вы, обычно, усиливаете отладчик для вашего кода применением ICE.

4.1 Точки останова выполнения

Точки останова выполнения – это то, что остановит выполнение программы, когда адрес программной памяти, на котором она установлена, будет достигнут. Например, если вы отлаживаете PIC среднего класса, и хотите остановить выполнение при прерывании, вы должны задать точку останова в программной памяти по адресу 0x04:

```
gpsim> break e 4
```

(Более точно, прерывание не должно обнаружится, поскольку эта точка останова будет вычислена – код обработки должен тоже перейти сюда).
Точка останова обнаруживается до выполнения инструкции. Другие симуляторы, как MPLAB, останавливаются после того, как инструкция выполнена. Во многих случаях эта разница не существенна. Однако если остановка задана на инструкциях 'goto' или 'call', тогда предпочтительнее сделать остановку до того, как обнаружится ветвление программы. Этим путем легче определить место ветвления.

4.1.1 Точки останова неправильных инструкций

gpsim автоматически будет останавливать выполнение программы при попытке выйти за ее пределы. Расположение памяти программы, которое не определено вашим исходным кодом, будет инициализировано, как неправильная инструкция ('Invalid instruction'). Это хорошо видно, когда вы дизассемблируете программу.

4.2 Регистровые точки останова

gpsim предоставляет возможность останавливаться при обращении к регистру, как для чтения, так и для записи, или в обоих случаях. Более того, возможно останавливаться при специфическом значении читаемом или записываемом в регистр.

4.3 Цикловые точки останова

Цикловые точки останова позволяют остановить программу определенной цикловой инструкцией. Представьте, что вы имеете 20 МГц PIC и хотите остановить программу через одну секунду симуляции. Вы должны задать точку останова инструкцией в 5 миллионов циклов. (При 4 циклах на выполнение одной инструкции. В дальнейшем предполагается ввести в gpsim возможность задавать инструкцию точки останова в терминах секунд).

Часть 5

Трассиров ка: Что случилось ?

Проверка текущего состояния вашей программы временами недостаточна для выявления причины ошибки. Часто полезным оказывается выяснить условия при которых индикатор переходит в текущее состояние. gpdim предоставляет историю или трассировку всего, что происходит – хотите вы этого, или нет – чтобы помочь в диагностике этих, другими методами плохо анализируемых, ошибок.

Что трассируется	***Примечания***
Счетчик программы (PC)	Адрес выполнения
Инструкции	opcode (код операции)
Чтение регистра	Значение и положение
Запись регистра	Значение и положение
Счетчик циклов	Текущее значение
Пропущенные инструкции	Пропущенные адреса
Статус регистра	В течение потенциальной модификации
Прерывания	
Точки останова	Тип
Сбросы	Тип

Команда 'trace' будет записывать содержимое буфера трассировки.
Большой замкнутый буфер (чей размер предустановлен) сохраняет информацию для буфера трассировщика. Когда он заполняется, он возвращается к началу и записывает поверх старой записи. Содержимое буфера трассировщика разбирается по фреймам, где один фрейм соответствует циклу симуляции.
Вот пример вывода трассировки:

```
gpsim> trace
0x00000000000025F6 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 0x000000C
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
```

Каждый фрейм начинается с нового цикла симуляции. Обычно это включается в инструкцию симуляции. Вот каждое из полей:

```
64-bit simulation cycle     processor     PC        opcode      instruction
0x00000000000026F6          16f452        0x001C    0x1003      iorwf reg3,w,0
```

Другие события, обнаруженные в процессе трассировочного фрейма, вырезаны. Обычно это будут трассировки чтения или записи регистра. Трассировки чтения показывают читаемое значение. Трассировки записи показывают записываемые значения и предыдущие значения регистра.

Часть 6
Симуляция реального мира: стимулы (воздействия)

Стимулы крайне полезны, если не необходимы, для симуляции. Они предоставляют значения для симуляционного взаимодействия с реальным миром.
Возможности стимулов gpsim разработаны точными, эффективными и гибкими. Модели для выводов ввода-вывода PIC имитируют реальные устройства. Например, открытый коллектор вывода порта A PIC16C84 может быть только в низком состоянии. Множество выводов ввода-вывода могут соединяться между собой так, что открытый коллектор порта A может получить “подтягивающий” резистор порта B. В первую очередь стимулы только обнаруживают, когда стимул меняет состояние. Другими словами, стимулы не выбирают определения их статуса.
Аналоговые стимулы тоже допустимы. Возможно создать ссылки на напряжение и уровни напряжения для симуляции, практически, любого рода вещей реального мира. Например, возможно комбинировать два аналоговых стимула вместе, чтобы создать сигналы подобные DTMF тонам.

6.1 Как они работают

В простейшем случае стимул действует как источник для вывода ввода-вывода PIC. Например, вы можете захотеть симулировать часы и измерять их период, используя TMR0 (таймер 0). В этом случае стимулом будет источник и TMR0 входной вывод PIC, как цепь. В gpsim вы создадите стимул для часов, используя команды стимула и соединив их с выводом ввода-вывода, используя команду узла.
В общем, вы можете иметь несколько “источников” и несколько “цепей”, которые соединены с узлами. (Хотя gpsim в данный момент ограничена “одно-портовыми” устройствами. Другими словами, она принимает, что земля обслуживается, как общая ссылка для источников и цепей). Хорошим аналогом будет цепь spice. spice список цепей (netlist) обращается к списку узлов в gpsim и элементы spice обращаются к источникам и цепям стимулов. Этот главный подход делает возможным создавать разнообразнейшее окружение симуляции. Вот список разных путей, которыми стимулы могут быть соединены:

1. Стимул соединен с одним I/O выводом
2. Стимул соединен с несколькими I/O выводами
3. Несколько стимулов присоединены к одному I/O выводу

4. Несколько стимулов соединены с несколькими I/O выводами
5. I/O вывод соединен с I/O выводом

Основная техника для реализации стимулов следующая:

1. Определить стимулы или стимул.
2. Определить узел.
3. Присоединить стимулы к узлам.

Чаще бывает, чем нет, определение стимула в файле.

6.1.1 **Соединения между стимулами**

Одной из проблем с этим узловым подходом к моделированию стимулов является возможность для соединения существующих. Например, если два I/O вывода соединены один с другим и работают в противоположных направлениях, соединение получится. gpsim принимает соединение с суммой атрибутов. Каждый стимул, даже если это ввод, производит влияние на узел. Это влияние заданной силы. Когда узел обновляется, gpsim будет просто объединять силу всех стимулов вместе и применять это числовое значение к узлу. Сила со значением нуль относится к отсутствию цепи. Большая положительная сила используется стимулом для положительного движения узла, большая отрицательная сила используется для отрицательного движения. Суммирование атрибутов полезно для “подтягивающих” резисторов. В примере с открытым коллектором порта A/ порт B слабое “подтягивание”, gpsim задает относительно слабую силу “подтягивания” для резисторов и большую отрицательную силу для открытого коллектора, если он активен, или не задает силу, если нет движения (изменения). Эффект конденсатора (пока не поддерживается) может быть симулирован с динамически меняющимся значением силы.

6.2 **Выводы I/O**

gpsim моделирует выводы I/O, как стимулы. Таким образом, где бы стимулы не использовались, I/O выводы могут быть заменены. Например, вы можете захотеть присоединить два I/O вывода один к другому, как присоединялся подтягивающий резистор порта B к порту A с открытым коллектором. gpsim автоматически создает стимул I/O вывода при создании процессора. Все, что

вам нужно, так это обозначить узел и затем назначить стимул для него. Имена для этих стимулов формируются связыванием имени порта с позицией бита I/O вывода. Например, бит 3 порта B называется portb3.
Вот список поддерживаемых типов стимулов выводов I/O:

Тип I/O выводов	***Функция***
INPUT_ONLY (только ввод)	Принимает только ввод (подобно MCLR)
BI_DIRECTIONAL (двунаправленные)	Может быть источником или цепью (большинство выводов I/O)
BI_DIRECTIONAL_PU (двунаправленные PU)	PU – подтягивающие резисторы (Порт B)
OPEN_COLLECTOR (открытый коллектор)	Могут двигаться только вниз (RA4 у c84)

Нет специального типа выводов для аналоговых выводов I/O. Все аналоговые вводы PIC мультиплексируются с цифровыми вводами. Определение I/O выводов будет всегда для цифровых вводов. gpsim автоматически определяет, если I/O вывод – аналоговый ввод.

6.3 Асинхронные стимулы

Асинхронные стимулы – аналоговые или цифровые стимулы, которые могут изменять состояние в любой заданный момент (ограниченный разрешением счетчика циклов). Они могут также определяться как повторяющиеся.

Параметр	***Функция***
start_cicle (начать цикл)	Количество циклов перед началом стимула
cycles[] (циклов)	Массив номеров циклов
data[] (данные)	Состояние стимула для цикла
period (период)	Количество циклов в одном периоде
initial_state (начальное состояние)	Начальное состояние перед data[0]

Когда стимул впервые инициализируется, он будет приведен в движение в 'initial state' (начальном состоянии) и будет оставаться в нем, пока CPU счетчик команд не станет равен 'start' (начать) циклу. После этого два массива cycles[] и data[] определяют выход стимула. Размер массивов тот же, что относится к

количеству событий, которые создавались. Таким образом количество событий, если было, обслуживает индекс этих двух массивов. Массив cycles[] определяет, когда событие обнаруживается, тогда как массив data[] определяет, какое состояние стимула будет введено. cycles[] измеряется в согласии с циклом start. Асинхронные стимулы могут быть сделаны периодическими через задание числа циклов в параметре period.

Вот пример, который генерирует три импульса, а затем повторяется:

```
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 **Аналоговый асинхронный стимул**

Аналоговый асинхронный стимул идентичен синхронным стимулам, исключая тот момент, что данные точки – это числа с плавающей точкой.

Модули

gpsim был разработан для отладки микропроцессоров. Однако микропроцессоры всегда часть системы. И неизменной часто встречающейся ошибкой становится интерфейс с системой. Модули предоставляют пользователю путь расширения gpsim и симуляции системы. Например, *системой* может быть процессор с несколькими подтягивающими резисторами и переключателями, или может быть процессор с LCD дисплеем. gpsim предоставляет несколько модулей, каждый может использоваться как для отладки, так и в качестве шаблона для создания новых модулей.

7.1 Модули gpsim

gpsim предоставляет следующие модули:

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

7.2 Написание новых модулей

Модули – это библиотека кодов. В Windows библиотека это dll, а в Unix общие библиотеки. Есть несколько деталей, которых модуль должен

придерживаться, но в основном модуль должен иметь полный доступ к gpsim API.

Часть 8

Символьная отладка

gpsim поддерживает таблицу символов.

Часть 9

Макросы

Часть 10

Hex файлы

Целевой код, симулируемый gpsim может содержаться в hex файле, или более точно в Intel Hex файле. gpsim принимает формат hex, предоставляемый программами gpasm и mpasm. HEX файл не предоставляет никакой символьной информации. Рекомендуется, чтобы hex файлы использовались только, если

1. вы обнаруживаете проблемы с использованием .cod файла, генерируемого вашим ассемблером или компилятором, или
2. ваш ассемблер или компилятор не генерирует .cod файлы.

Также вы должны запастись процессором, когда загружаете hex файл. Посмотрите команду load.

ICD

gpsim поддерживает (частично) первую версию ICD (в противоположность ICD2 (с формой хоккейной шайбы))

Специальная конфигурация кода

Прочитайте руководство пользователя по MPLAB ICD.
Вот краткая версия:

невозможно по крайней мере: обнаружение неопределенного выхода, низковольтное программирование и полная защита кода. Возможно лучше выключать также watchdog. Обратитесь к руководству пользователя по MPLAB ICD за более подробной информацией.
иметь NOP в качестве первой инструкции.
Не трогайте RB6 или RB7.
Не используйте последний уровень стека.
Не используйте следующие регистры и программные слова:

Процессор	***Регистр***	***Программа***
-870/1/2	0x79,0xBB-0xBF	0x6E0-Ox7FF
-873/4	0x6D, 0x1FD, 0xEB-0xF0, 0x1EB-Ox1F0	0xEE0-0xFFF
-876/7	0x70, 0x1EB-0x1EF	0x1F00-0x1FFF

icdprog

Загрузите и установите icdprog.
Используйте icdprog для программной цели с hex файлом (icdprog mycode.hex).

Использование ICD

Запустите gpsim, как указано ниже:
gpsim -d /dev/ttyS0 -s mycode.cod

принимается, что ICD присоединено к первому COM-порту.
Теперь вы можете напечатать icd, чтобы получить информацию:
***gpsim> icd*
ICD version “2.31.00” was found.
Target controller is 16F877 rev 13.
Vdd: 5.2 Vpp: 13.3
Debug module id present

2.13 - это версия внутреннего программного обеспечения. Я только попробовал эту частную версию...

Вы можете шагать, сбросить, запустить, остановить, задать точки останова и прочитать файловые регистры. Это работает как в командной строке, так и с графической оболочкой.

ICD TODO

MPLAB имеет установку для частоты целевого CPU, я только попробовал с 20 МГц кристаллом, так что возможно придется согласовывать тайм-аут последовательного порта с установками gpsim.
Окна исходного кода, дизассемблера, наблюдения, символьное и RAM работают. А остальные нет. Я полагал, что макет должен быть работоспособен хотя бы для PIC, но этого нет.
EEPROM поддерживается
Модифицируются данные
Зафиксировано, UI должно давать больше сигналов обратной связи в части, что случилось во время длинных задержек.
Лучшее обнаружение ошибок. gpsim не всегда видит, что цель не функциональна.

Теория операций

Этот раздел предназначен только для тех, кого может заинтересовать, как gpsim оперирует. Информация здесь максимально точна. Однако, поскольку gpsim развивается, тоже должны делать детали теории операций. Используйте информацию, предложенную здесь, как введение высокого уровня и используйте (хорошо закомментировано :]) исходный текст, чтобы выяснить детали.

12.1 Задний план

gpsim написан в основном на C++. Почему? Хорошо, основным соображением была легкость разработки иерархической модели PIC. Если вы подумываете о микроконтроллере, так действительно легко конструировать различные компоненты. C++ сам предоставляет все для этой концептуализации. Более того, Microchip, как и другие производители микроконтроллеров, создал семейство устройств, которые очень похожи одно на другое. Опять же, C++ предоставляет “наследование”, которое допускает, чтобы взаимосвязь была обобщена на различные модели PIC.

12.2 Инструкции

Представлен базовый класс для 14-битовых инструкций (Я планирую сделать в будущем следующий шаг и создать базовый класс, из которого можно получить все PIC инструкции). Он в первую очередь служит двум целям: хранить все общее для каждой инструкции и дать значение для виртуальным функциям основного доступа. Общая информация состоит из имени – или более точно, мнемоники инструкции, opcode, и указатель на процессор, пользующийся инструкцией. Некоторые из виртуальных функций выполняемые и именованные. После декодирования hex файла содержимое инструкций создается и сохраняется в массиве, названном program_memory. Индексом массива служит адрес, по которому пребывает инструкция. Для выполнения инструкции вызывается следующая последовательность кодов:

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

которая говорит, взять инструкцию по текущему состоянию счетчика команд (pc.value) и вызвать через виртуальную функцию execute(). Такой подход позволяет, чтобы точки останова при выполнении были легко заданы. Инструкции специальных точек останова могут замещать соответствующие, находящиеся в массиве памяти программы. Когда вызывается execute, может появиться точка останова.

12.3 Основные регистры файла

Регистр файла симулируется классом file_register. Есть один образец объекта file_register для каждого регистра файла PIC. Все регистры собраны вместе в массив, названный register, который индексируется регистрами, обращающимися к адресам PIC. Массив линейный и не разбит на банки, как в реальном PIC. (Использование банков памяти поддерживается при симуляции).

12.4 Специальные файловые регистры

Специальные файловые регистры все остальные регистры, которые не относятся к основным файловым регистрам. Это включает регистры ядра, как status и option, а также периферийные регистры, как eeadr для eeprom. Специальные файловые регистры производные от основных файловых регистров и также хранятся в массиве register. Есть один образец для каждого регистра – даже, если регистр доступен более, чем в одном банке. Так, например, есть один образец для регистра status, хотя он может быть доступен через массив register более, чем в одном месте.
Все файловые регистры доступны через виртуальные функции put и get. Это сделано по двум основным причинам. Во-первых, это удобно для инкапсуляции перезаписи точек останова (для регистровых точек останова) в файловые регистры, а не в инструкции. Во-вторых, что более важно, допускают производные классы для реализации put и get более специфично. Например, put для indf регистра имеет целый ряд отличий от put для intcon регистра. В каждом случае put инициирует действия между просто сохраняемыми байтами данных в массиве. Это также позволяет использовать следующую последовательность кодов для легкой реализации:

```
movlw    trisa          ;   получает адрес trisa
movwf     fsr
movf      indf,w        ;    читает trisa косвенно
```

12.5 Пример инструкции

Вот пример кода для инструкции movf, который иллюстрирует то, что обсуждалось выше. Где-нибудь в gpsim выполняется последовательность кодов:

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

Давайте скажем, что PC указывает на инструкцию movf. Виртуальная фукнкция ->execute() вызвет MOVF::execute. Я добавил особые комментарии (их нет в основном коде) для детальной иллюстрации того, что происходит.

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

// Все инструкции traced (обсуждается ниже). Достаточно
// только сохранить opcode. Однако даже этого может оказаться
// не надо, если счетчик команд тоже трассируется. Ожидается,
// что это исчезнет в дальнейшем...
trace.instruction(opcode);

// 'source' – это указатель на объект 'file_register'. Он
// инициализируется чтением массива 'registers'. Заметьте, что
// индекс зависит от бит 'rp' (в действительности от одного) в
// регистре status. Время сохраняется кэшированием rp в
// противоположность декодированию регистра status.
source = cpu->registers[cpu->rp |
opcode&REG_IN_INSTRUCTION_MASK];

// У нас нет соображений по тому, до какого регистра мы пытаемся
```

```
// добраться и как он будет достигнут или есть ли на нем
// точка останова. Нет проблем, виртуальная функция 'get'
// разрешит все эти детали и “сделает все правильно”
source_value = source->get();

// Если мы обращаемся к W, тогда конструктор уже
// инициализировал 'destination'. Иначе source и destination
// те же самые.
if(opcode&DESTINATION_MASK)
  destination = source;   // Результат отправится в source

// Запишем значение sorce в destination. Вновь у нас нет
// соображений, где может быть destination или
// как данные будут там записаны.
destination->put(source_value);

// Инструкция movf установит Z (ноль) бит в регистре status,
// если значение sorce было нулем.
cpu->status.put_Z(0==source_value);

// Окончательно, наращиваем PC на единицу.
cpu->pc.increment();
}
```

12.6 Трассировка

Все, что симулируется, трассирует -all все время. Буфер трассировки - единый огромный циклический буфер целых. Информация после OR с символами трассировки сохраняется в буфере трассировки. Не делается попыток ассоциировать пункты в буфере трассировщика, пока симулятор симулирует PIC. Хотя, если вы взглянете на строки буфера, то увидите нечто похожее: cycle counter = ..., opcode fetch = ..., register read = ..., registar write = ..., и т. д. Однако эта информация – пост-процесс для того, чтобы удостовериться, что происходит и когда оно происходит. Можно использовать эту информацию для “undo” симуляции, другими словами, вы можете сделать шаг назад. Хотя я не разрабатывал этого совсем.

12.7 Точки останова

Точки останова можно разбить на три категории: выполнения, регистровые и циклов.

Выполнения:

Для точек останова выполнения специальная инструкция, соответственно названная 'Breakpoint_instruction', создана и размещена в массиве программной памяти в месте, где желательна точка останова. Оригинальная инструкция сохраняется во вновь открытой инструкции точки останова. Когда точка останова очищается, оригинальная инструкция извлекается из инструкции точки останова и помещается назад в массив программной памяти.
Заметьте, что эта схема имеет нулевую перезапись. Симулятор будет потревожен только тогда, когда встречается точка останова.

Регистровые:

Есть, по крайней мере, четыре разных типа точек останова, которые могут применяться к регистрам: чтение любого значения, запись любого значения, чтение определенного значения, запись определенного значения. Подобно точкам останова выполнения, есть специальные регистровые точки останова, которые замещают объект регистра. Так что, когда пользователь устанавливает точку останова на регистр 0x20, например, создается новый объект точки останова и вставляется в массив файла регистров в позицию 0x20. Когда симулятор пытается получить доступ к регистру по адресу 0x20, вместо него получается доступ к объекту точки останова.
Заметьте, что эта схема также имеет нулевую перезапись, активизируясь, когда встречается точка останова.

Цикловые:

Цикловые точки останова позволяют gpsim изменять выполнение в обозначенных циклах инструкций. Это полезно для выполнения вашей

симуляции на определенные промежутки времени. Внутренне gpsim применяет широкое использование цикловых точек останова. Например, объект TMR0 может программироваться на генерацию периодических цикловых точек останова.

Цикловые точки останова имплементируются с сортированным списком двойных связей. Список связей содержит две части информации (кроме связей): цикл, на котором обнаруживается остановка, и функция возврата из вызова (Функция возврата из вызова – указатель на функцию. В этом контексте gpsim получает указатель на функцию, которая должна появиться, вызванная при обнаружении циклового останова), которая должна появится, когда обнаруживается цикл. Логика останова крайне проста. Когда счетчик циклов наращивается (инкрементируется), он сравнивается со следующей желаемой цикловой точкой останова. Если НЕТ совпадения, тогда инкрементируем. Так что перезапись для цикловых остановок требует времени для реализации и сравнения. Если совпадение ЕСТЬ, тогда функция возврата из вызова ассоциируется с появлением этой точки останова в двух-связном списке обслуживания, как ссылка на следующую цикловую остановку.