Отладка с помощью GDB

Отладчик GNU уровня исходного кода

Восьмая Редакция, для GDB версии 5.0
Март 2000

Ричард Столмен, Роланд Пеш, Стан Шебс и другие.

(Присылайте сообщения об ошибках и комментарии к GDB по адресу bug-gdb@gnu.org.)
Отладка с помощью GDB
TEXinfo 1999-10-01.07

Published by the Free Software Foundation
59 Temple Place - Suite 330,
Boston, MA 02111-1307 USA
ISBN 1-882114-77-9

Оглавление

Обзор GDB

Назначение отладчика, такого как GDB—позволить вам увидеть, что происходит "внутри" другой программы во время ее выполнения, или что делала другая программа в момент краха.

GDB может выполнять действия четырех основных типов (а также другие, поддерживающие эти основные), чтобы помочь вам выявить ошибку:

- Начать выполнение вашей программы, задав все, что может повлиять на ее поведение.
- Остановить вашу программу при указанных условиях.
- Исследовать, что случилось, когда ваша программа остановилась.
- Изменить вашу программу, так что вы можете экспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

Вы можете использовать GDB для отладки программ, написанных на Си и Си++. Для получения более подробной информации, смотрите Раздел 9.4 [Поддерживаемые языки], с. 81. Для дополнительной информации, смотрите Раздел 9.4.1 [Си и Си++], с. 81.

GDB частично поддерживает языки Модула-2 и Chill. Для получения информации о Модуле-2, смотрите Раздел 9.4.2 [Модула-2], с. 87. Для получения информации о Chill, см. Раздел 9.4.3 [Chill], с. 91

Отладка программ на Паскале, которые используют множества, поддиапазоны, файловые переменные или вложенные функции, в настоящее время не работает. GDB не поддерживает ввод выражений, вывод значений, и аналогичные возможности, использующие синтаксис Паскаля.

GDB может использоваться для отладки программ, написанных на Фортране, хотя может возникнуть необходимость ссылаться на некоторые переменные с использованием знака подчеркивания на конце.

Свободно распространяемые программы

GDB—*свободная программа*, защищенная Универсальной Общественной Лицензией GNU (GPL). GPL предоставляет вам свободу копировать или изменять программу, но каждый человек, получая копию, также получает свободу изменять эту копию (это означает, что он должен получить доступ к исходному коду), и свободу распространять последующие копии. Обычные компании, разрабатывающие программы, используют авторские права для ограничения ваших свобод; Фонд Свободного Программного Обеспечения использует GPL для сохранения этих свобод.

Главное, Универсальная Общественная Лицензия—это лицензия, говорящая, что вы имеете эти свободы и что вы не можете их у кого-либо отнять.

Кто внес вклад в развитие GDB

Первоначальным автором GDB, как и многих других программ GNU, был Ричард Столмен. Многие другие люди внесли вклад в его разработку. Этот раздел пытается отдать должное основным участникам разработки. Одним из достоинств свободных программ является то, что любой человек имеет свободу делать вклад в их развитие; к сожалению, мы не можем в действительности поблагодарить здесь всех. Файл '`ChangeLog`' в поставке GDB представляет детальнейший отчет.

Изменения, сделанные задолго до версии 2.0 потеряны в тумане времен.

> *Оправдание:* Дополнения к этому разделу особенно приветствуются. Если вы или ваши друзья (или враги, чтобы соблюдать справедливость) были незаслуженно пропущены в этом списке, мы будем рады добавить ваши имена!

Особенно мы хотим сказать спасибо тем, кто присматривал за GDB между основными выпусками, чтобы они не сочли свои многочисленные заслуги оставленными без благодарности: Эндрю Кагни (выпуск 5.0); Джим Бленди (выпуск 4.18); Джесон Моленда (выпуск 4.17); Стан Шебс (выпуск 4.14); Фред Фиш (выпуски 4.16, 4.15, 4.13, 4.12, 4.11, 4.10, и 4.9); Сту Гроссман и Джон Гилмор (выпуски 4.8, 4.7, 4.6, 4.5, и 4.4); Джон Гилмор (выпуски 4.3, 4.2, 4.1, 4.0, и 3.9); Джим Кингдон (выпуски 3.5, 3.4, и 3.3) и Ренди Смит (выпуски 3.2, 3.1, и 3.0).

Ричард Столмен, которому в различные времена помогали Петер ТерМаат, Крис Хенсон и Ричард Млинарик, занимался выпусками до 2.8.

Михаэль Тиманн является автором большей части поддержки GNU Си++ в GDB, со значительным дополнительным вкладом от Пера Бозера. Джеймс Кларк написал дешифровщик[1] имен GNU Си++. Ранняя работа по Си++ была сделана Петером ТерМаатом (который также сделал много общей работы по обновлению, приведшей к выпуску 3.0).

Для исследования многих форматов объектных файлов, GDB 4 использует библиотеку подпрограмм BFD. BFD был совместным проектом Дэвида В. Хенкел-Волласа, Рича Пиксли, Стива Чемберлена, и Джона Гилмора.

Дэвид Джонсон первоначально написал поддержку COFF; Пейс Виллисон первоначально сделал поддержку инкапсулированного COFF.

Брент Бенсон из Harris Computer Systems сделал поддержку DWARF 2.

Адам ДеБур и Брэндли Дэвис сделали поддержку ISI Optimum V. Пер Бозер, Нобоюки Хикичи, и Алессандро Форин сделали поддержку MIPS. Жан-Даниэль Фекет сделал поддержку Sun 386i. Крис Хенсон улучшил поддержку HP9000. Нобоюки Хикичи и Томоюки Хаси сделали поддержку Sony/News OS 3. Дэвид Джонсон сделал поддержку Encore Umax. Юрки Куоппала сделал поддержку Altos 3068. Джефф Ло сделал поддержку HP PA и SOM. Кейс Паккард сделал поддержку NS32K. Доуг Ребсон сделал поддержку Acorn Risc Machine. Боб Раск сделал поддержку Harris Nighthawk CX-UX. Крис Смит сделал поддержку Convex (и отладку программ на Фортране). Джонатан Стоун сделал поддержку Pyramid. Михаэль Тиманн сделал поддержку SPARC. Тим Такер сделал поддержку для Gould NP1 и Gould Powernode. Пейс Виллисон сделал поддержку Intel 386. Джей Восбург сделал поддержку Symmetry.

---

[1] В Си++ и других объектно-ориентированных языках программирования, у вас может быть несколько функций с одним именем, но с аргументами разных типов. Например:

```
int add_two(int a, int b);
double add_two(double a, double b);
double add_two(double a, int b);
```

Компилятор генерирует код для вызова верной функции по заданным аргументам. Это называется *перегрузкой функций*.

Однако, компоновщик требует, чтобы все символы имели недвусмысленные имена. Поэтому компилятор *шифрует* (от английского “mangle”) имена перегруженных функций так, чтобы они включали типы аргументов и возвращаемые значения. К примеру, приведенные выше функции могут быть могут быть зашифрованы примерно так:

```
add_two_Ret_int_int_int
add_two_Ret_double_double_double
add_two_Ret_double_double_int
```

(в действительности шифрованные имена выглядят безобразнее). В результате компоновщик может правильно обработать перегруженные функции. *Дешифровщик* (от английского “demangler”)—это программа (или функция), которая выполняет обратную операцию: анализируя зашифрованные имена, она выдает исходную сигнатуру функции. Это необходимо, чтобы отобразить то, что программист сможет понять и связать с исходным текстом своей программы. Для этого любому средству отладки, поддерживающему Си++, обычно требуется дешифровщик. *(Прим. переводчика)*

Андреас Шваб сделал поддержку M68K Linux.

Рич Шаефер и Петер Шауер помогли реализовать поддержку разделяемых библиотек SunOS.

Джей Фенласон и Роланд МакГрес проверили совместимость GDB и GAS по нескольким наборам машинных инструкций.

Патрик Дювал, Тед Голдстейн, Викрам Кока и Гленн Инжел помогли разработать удаленную отладку. Intel Corporation, Wind River Systems, AMD и ARM сделали модули удаленной отладки для целей i960, VxWorks, A29K UDI, и RDI соответственно.

Брайан Фокс является автором библиотек Readline, предоставляющих историю команд и возможность редактирования командной строки.

Эндрю Бирс из SUNY Buffalo написал код для переключения языков, поддержку Модулы-2, главу 'Языки' этого руководства.

Фред Фиш написал большую часть поддержки Unix System Vr4. Он также улучшил поддержку завершения команд для поддержки перегруженных символов Си++.

Hitachi America, Ltd. спонсировала поддержку для процессоров H8/300, H8/500 и Super-H.

NEC спонсировала поддержку процессоров v850, Vr4xxx и Vr5xxx.

Mitsubishi спонсировала поддержку процессоров D10V, D30V и M32R/D.

Toshiba спонсировала поддержку процессора TX39 Mips.

Matsushita спонсировала поддержку процессоров MN10200 и MN10300.

Fujitsu спонсировала поддержку процессоров SPARClite и FR30.

Кунг Шу, Джефф Ло и Рик Слэдки добавили поддержку аппаратных точек наблюдения.

Михаэль Снайдер добавил поддержку точек трассировки.

Сту Гроссман написал gdbserver.

Джим Кингдон, Петер Шауер, Ян Тейлор и Сту Гроссман сделали почти бесчисленное количество исправлений и улучшений во всем GDB.

Следующие люди из Hewlett-Packard Company сделали поддержку архитектуры PA-RISC 2.0, HP-UX 10.20, 10.30 и 11.0 (усеченный режим), реализации нитей HP в ядре, компилятора HP aC++, и конечного интерфейса пользователя: Бен Крепп, Ричард Тайтл, Джон Бишоп, Сюзан Макчиа, Кэси Манн, Сэтиш Пай, Индиа Поул, Стив Рейраур и Елена Заннони. Ким Хаас предоставил специфичную для HP информацию для этого руководства.

Cygnus Solutions спонсировала поддержку GDB и большую часть его развития с 1991 года. Среди инженеров Cygnus, работавших над GDB на постоянной основе, Марк Александер, Джим Бленди, Пер Бозер, Кевин Беттнер, Эдит Эпштейн, Крис Фейлор, Фред Фиш, Мартин Хант, Джим Ингам, Джон Гилмор, Сту Гроссман, Кунг Шу, Джим Кингдон, Джон Мецлер, Фернандо Нассер, Джеффри Ноер, Дон Перчик, Рич Пиксли, Зденек Радуч, Кейс Сейц, Стан Шебс, Дэвид Тейлор и Елена Заннони. Кроме того, Дейв Бролли, Ян Кармихаэль, Стив Чемберлен, Ник Клифтон, Джэй Ти Конклин, Стен Кокс, Ди Джей Делори, Ульрих Дреппер, Фрэнк Эйглер, Дуг Эванс, Син Фаган, Дэвид Хенкель-Воллас, Ричард Хендерсон, Джефф Холком, Джефф Ло, Джим Лемке, Том Лорд, Боб Мансон, Михаэль Мейсснер, Джейсон Меррилл, Кэтрин Мур, Дрю Мосли, Кен Робурн, Гавин Ромиг-Кох, Роб Савой, Джейми Смит, Майк Стамп, Ян Тейлор, Анжела Томас, Михаэль Тиманн, Том Тромей, Рон Унро, Джим Вилсон и Дэвид Зун также сделали свой вклад в большей или меньшей степени.

1 Пример сеанса GDB

Вы можете пользоваться этим руководством в свое удовольствие, чтобы прочитать о GDB все. Однако, достаточно небольшого количества команд, чтобы начать пользоваться отладчиком. Эта глава иллюстрирует эти команды.

В этом примере сеанса мы выделяем ввод пользователя так: **ввод**, чтобы его было проще отличить от находящегося рядом вывода программы.

В одной из предварительных версий программы GNU `m4` (настраиваемый макропроцессор), была допущена следующая ошибка: иногда, при замене строк, определяющих кавычки, со значений по умолчанию, команды, использовавшиеся для поиска одного макроопределения внутри другого, прекращали работать. В следующем коротком сеансе `m4`, мы определим макрос `foo`, который расширяется до `0000`; затем мы используем встроенную процедуру `m4 defn`, чтобы определить `bar` точно также. Однако, когда мы изменим открывающую кавычку на `<QUOTE>`, а закрывающую на `<UNQUOTE>`, та же самая процедура не сможет определить новый синоним `baz`:

```
$ cd gnu/m4
$ ./m4
define(foo,0000)

foo
0000
define(bar,defn(‘foo’))

bar
0000
changequote(<QUOTE>,<UNQUOTE>)

define(baz,defn(<QUOTE>foo<UNQUOTE>))
baz
C-d
m4: End of input: 0: fatal error: EOF in string
```
[1]

Попытаемся с помощью GDB понять, что же происходит.

```
$ gdb m4
GDB is free software and you are welcome to distribute copies
 of it under certain conditions; type "show copying" to see
 the conditions.
There is absolutely no warranty for GDB; type "show warranty"
 for details.

GDB 5.0, Copyright 1999 Free Software Foundation, Inc...
(gdb)
```

GDB читает только минимум символьных данных, достаточный для того, чтобы знать, где в случае необходимости искать остальные; в результате первое приглашение появляется очень быстро. Теперь мы велим GDB использовать меньшую ширину экрана, чем обычно, чтобы примеры умещались на страницах этого руководства.

```
(gdb) set width 70
```

Нам необходимо увидеть, как работает встроенная процедура `m4 changequote`. Посмотрев исходный текст, мы знаем, что соответствующей подпрограммой является `m4_changequote`, так что мы устанавливаем там точку останова с помощью команды GDB `break`.

[1] m4: Конец ввода: 0: фатальная ошибка: EOF в строке *(Прим. переводчика)*

```
(gdb) break m4_changequote
Breakpoint 1 at 0x62f4: file builtin.c, line 879.
```

Используя комманду `run`, мы запускаем `m4` под управлением GDB; до тех пор, пока управление не достигло подпрограммы `m4_changequote`, программа выполняется как обычно:

```
(gdb) run
Starting program: /work/Editorial/gdb/gnu/m4/m4
define(foo,0000)

foo
0000
```

Чтобы активировать точку останова, мы вызываем `changequote`. GDB приостанавливает выполнение `m4`, выводя на экран информацию о том месте, где он остановился.

```
changequote(<QUOTE>,<UNQUOTE>)

Breakpoint 1, m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:879
879         if (bad_argc(TOKEN_DATA_TEXT(argv[0]),argc,1,3))
```

Теперь мы используем команду `n` (`next`), чтобы продолжить выполнение до следующей строки текущей функции.

```
(gdb) n
882         set_quotes((argc >= 2) ? TOKEN_DATA_TEXT(argv[1])\
 : nil,
```

`set_quotes` выглядит как нужная нам подпрограмма. Мы можем войти в нее, используя команду `s` (`step`) вместо `next`. `step` переходит на следующую исполняемую строку *любой* подпрограммы, так что она входит в `set_quotes`.

```
(gdb) s
set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
530         if (lquote != def_lquote)
```

Отображение, показывающее подпрограмму (и ее параметры), где `m4` сейчас приостановлен, называется отображением кадра стека. Оно показывает обзор стека. Мы можем использовать команду `backtrace` (которая может сокращаться как `bt`), чтобы посмотреть, в каком месте стека мы находимся: команда `backtrace` выводит кадр стека для каждой активной подпрограммы.

```
(gdb) bt
#0  set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
#1  0x6344 in m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:882
#2  0x8174 in expand_macro (sym=0x33320) at macro.c:242
#3  0x7a88 in expand_token (obs=0x0, t=209696, td=0xf7fffa30)
    at macro.c:71
#4  0x79dc in expand_input () at macro.c:40
#5  0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

Чтобы посмотреть, что же происходит, мы проходим по шагам еще несколько строк. Первые два раза мы можем использовать ‘`s`’; следующие два раза мы пользуемся командой `n`, чтобы избежать вхождения в подпрограмму `xstrdup`.

```
(gdb) s
0x3b5c  532         if (rquote != def_rquote)
(gdb) s
```

```
0x3b80  535         lquote = (lq == nil || *lq == '\0') ?  \
def_lquote : xstrdup(lq);
(gdb) n
536         rquote = (rq == nil || *rq == '\0') ? def_rquote\
 : xstrdup(rq);
(gdb) n
538         len_lquote = strlen(rquote);
```

Последняя выведенная строка выглядит несколько странно; мы можем исследовать переменные `lquote` и `rquote`, чтобы понять, действительно ли это определенные нами новые левая и правая кавычки. Мы используем команду `p` (`print`), чтобы посмотреть их значения.

```
(gdb) p lquote
$1 = 0x35d40 "<QUOTE>"
(gdb) p rquote
$2 = 0x35d50 "<UNQUOTE>"
```

Действительно, `lquote` и `rquote` являются новыми кавычками. Для просмотра контекста, мы можем вывести десять строк исходного текста, окружающих текущую строку, используя команду `l` (`list`).

```
(gdb) l
533             xfree(rquote);
534
535         lquote = (lq == nil || *lq == '\0') ? def_lquote\
 : xstrdup (lq);
536         rquote = (rq == nil || *rq == '\0') ? def_rquote\
 : xstrdup (rq);
537
538         len_lquote = strlen(rquote);
539         len_rquote = strlen(lquote);
540     }
541
542     void
```

Давайте пройдем еще две строки, которые устанавливают `len_lquote` и `len_rquote`, и затем исследуем значения этих переменных.

```
(gdb) n
539         len_rquote = strlen(lquote);
(gdb) n
540     }
(gdb) p len_lquote
$3 = 9
(gdb) p len_rquote
$4 = 7
```

Это, очевидно, неправильно, так как `len_lquote` и `len_rquote` обозначают длины переменных `lquote` и `rquote` соответственно. Мы можем присвоить им лучшие значения, используя команду `p`, так как она может выводить значение любого выражения, а выражение может содержать вызовы подпрограмм и присваивания.

```
(gdb) p len_lquote=strlen(lquote)
$5 = 7
(gdb) p len_rquote=strlen(rquote)
$6 = 9
```

Достаточно ли этого, чтобы решить проблему использования новых кавычек во встроенной процедуре `m4 defn`? Мы можем продолжить выполнение `m4` командой `c` (`continue`), и затем попробовать пример, первоначально вызывавший ошибку:

```
(gdb) c
Continuing.

define(baz,defn(<QUOTE>foo<UNQUOTE>))

baz
0000
```

Успех! Теперь новые кавычки работают так же хорошо, как и стандартные. Кажется, проблема заключалась лишь в двух опечатках, приводивших к неправильному определению длин. Мы позволим `m4` выйти, подавая ему на вход EOF:

```
C-d
Program exited normally.
```

[2]

Сообщение '`Program exited normally.`' исходит от GDB; оно показывает, что `m4` закончил выполнение. Мы можем завершить наш сеанс работы с GDB командой `quit`.

```
(gdb) quit
```

[2] Программа завершилась нормально. *(Прим. переводчика)*

2 Вход и выход из GDB

Эта глава посвящена тому, как запустить GDB и как из него выйти. Основные принципы:

- введите ‘gdb’ для вызова GDB.
- введите quit или *C-d* для выхода из него.

2.1 Вызов GDB

Вызывайте GDB путем запуска программы gdb. Начав работу, GDB считывает команды с терминала до тех пор, пока вы не скажете ему выйти.

Вы также можете запустить gdb с различными аргументами и ключами, чтобы в самом начале лучше настроить среду отлаживания.

Ключи командной строки, описанные здесь, предназначены для охвата различных ситуаций; в дейтсвительности, в некоторых средах часть этих ключей может быть недоступна.

Чаще всего GDB вызывается с одним аргументом, который определяет исполняемую программу:

gdb *программа*

Вы также можете указать при старте как исполняемую программу, так и файл дампа памяти:

gdb *программа дамп*

Если вы хотите отладить выполняющийся в данный момент процесс, то вместо этого, вы можете указать вторым аргументом идентификатор этого процесса:

gdb *программа* 1234

присоединит GDB к процессу 1234 (если, конечно, у вас нет файла с именем ‘1234’, GDB сначала проверяет наличие файла дампа памяти).

Преимущества, которые можно получить при использовании второго аргумента командной строки, требуют наличия достаточно совершенной операционной системы; если вы используете GDB как удаленный отладчик, присоединенный к компьютеру без операционной системы, там вообще может не быть понятия “процесса”, и часто нет никакого способа получить дамп. GDB предупредит вас, если ему не удается присоединиться к процессу или считать файл дампа памяти.

Вы можете запустить gdb без вывода начального сообщения, описывающего отсутствие гарантии на него, указав -silent:

```
gdb -silent
```

Кроме того, вы можете контролировать процесс запуска GDB с помощью ключей командной строки. GDB может сам напомнить вам о доступных ключах.

Введите

```
gdb -help
```

чтобы вывести на экран все доступные опции с кратким описанием их использования (сокращенный эквивалент—‘gdb -h’).

Все заданные вами ключи и параметры командной строки обрабатываются последовательно. Порядок становится важным при использовании ключа ‘-x’.

2.1.1 Выбор файлов

При запуске, GDB считывает параметры, отличные от ключей, как указатели на исполняемую программу и файл дампа (или идентификатор процесса), точно так же, как если бы эти параметры задавались ключами '`-se`' и '`-c`' соответственно. (GDB считает первый параметр, не имеющий соответствующего флага ключа, эквивалентом ключа '`-se`', за которым следует этот параметр; а второй параметр, не имеющий соответствующего флага ключа, если он есть, эквивалентом ключа '`-c`', за которым следует этот параметр.)

Если GDB был сконфигурирован без включения поддержки файлов дампа, что имеет место для большинства встроенных целей, то он выразит недовольство вторым аргументом и проигнорирует его.

Многие ключи имеют как длинную, так и краткую формы; в следующем списке приводятся обе. GDB также распознает сокращения длинных форм, не являющиеся двусмысленными. (Вы можете, по желанию, обозначать ключи с помощью '`-`', а не '`-`', хотя мы показываем наиболее употребляемый формат.)

`-symbols` *файл*
`-s` *файл*
: Читать таблицу символов из файла *файл*.

`-exec` *файл*
`-e` *файл*
: Использовать *файл* как исполняемый для выполнения и исследования данных вместе с дампом памяти, когда это необходимо.

`-se` *файл*
: Читать таблицу символов из файла *файл* и использовать его как исполняемый файл.

`-core` *файл*
`-c` *файл*
: Использовать *файл* как дамп памяти для исследования.

`-c` *номер*
: Присоединиться к процессу с идентификатором *номер*, также, как по команде `attach` (при условии, что нет файла в формате дампа памяти с именем *номер*; в этом случае '`-c`' определяет этот файл как дамп для считывания).

`-command` *файл*
`-x` *файл*
: Выполнить команды GDB из файла *файл*. См. Раздел 16.3 [Командные файлы], с. 158.

`-directory` *каталог*
`-d` *каталог*
: Добавить *каталог* к путям поиска файлов с исходными текстами.

`-m`
`-mapped`
: *Предупреждение: этот ключ зависит от возможностей операционной системы, которые реализованы не везде.*
Если отображаемые в память файлы поддерживаются в вашей системе через системный вызов `mmap`, вы можете использовать этот ключ, чтобы GDB записывал символы из вашей программы в файл в текущем каталоге, допускающий повторное использование. Если программа, которую вы отлаживаете, называется '`/tmp/fred`', то отображаемым символьным файлом будет '`/tmp/fred.syms`'. Последующие отладочные сеансы GDB замечают наличие этого файла и могут быстро отобразить в память символьную информацию из него, а не читать таблицу символов из выполняемого файла.
Файл '`.syms`' специфичен для рабочей машины, на которой запускается GDB. Он содержит точный образ внутренней символьной таблицы GDB. Он не может быть разделен между несколькими рабочими платформами.

`-r`
`-readnow`
: Читать символьную таблицу каждого файла, содержащего таблицу символов, сразу целиком, а не стандартным образом, при котором она считывается посте-

пенно по мере необходимости. Эта команда замедляет запуск, но дальнейшие операции производятся быстрее.

Ключи `-mapped` и `-readnow` обычно используются вместе, чтобы построить файл '`.syms`', который содержит полную информацию о символах. (См. Раздел 12.1 [Команды для задания файлов], с. 105, для информации о файлах '`.syms`'.) Вот простой вызов GDB, не делающий ничего, кроме построения файла '`.syms`' для использования в будущем:

```
gdb -batch -nx -mapped -readnow имя-программы
```

2.1.2 Выбор режимов

Вы можете вызывать GDB в различных альтернативных режимах—например, в пакетном или в "тихом" режиме.

`-nx`
`-n` Не выполнять команды ни из каких файлов инициализации (обычно называемых '`.gdbinit`', или '`gdb.ini`' на PC). В нормальном режиме, GDB выполняет команды из этих файлов после обработки всех командных ключей и параметров. См. Раздел 16.3 [Командные файлы], с. 158.

`-quiet`
`-silent`
`-q` "Тихий". Не печатать вводное сообщение и информацию об авторских правах. Эти сообщения также подавляются в пакетном режиме.

`-batch` Выполняться в пакетном режиме. Выйти со значением `0` после обработки всех командных файлов, заданных ключем '`-x`' (и всех команд из инициализационных файлов, если это не запрещено ключем '`-n`'). Выйти с ненулевым значением, если во время выполнения команд GDB из командных файлов произойдет ошибка.

Пакетный режим может быть полезен при вызове GDB как фильтра; например, чтобы загрузить программу и запустить ее на другом компьютере. Для того, чтобы сделать это более удобным, сообщение

```
Program exited normally.
```

(которое обычно выдается при завершении программы, выполняемой под управлением GDB), при выполнении в пакетном режиме не выдается.

`-nowindows`
`-nw` "Без окон". Если GDB имеет встроенный графический интерфейс пользователя (GUI), то этот ключ велит GDB использовать только интерфейс командной строки. Если GUI недоступен, этот ключ не оказывает никакого действия.

`-windows`
`-w` Если GDB включает GUI, этот ключ требует использовать его, если только возможно.

`-cd` *каталог*
Запустить GDB, используя в качестве рабочего каталога *каталог*, вместо текущего.

`-fullname`
`-f` GNU Emacs устанавливает этот ключ, когда вызывает GDB как подпроцесс. Это велит GDB выводить полное имя файла и номер строки в стандартном, распознаваемом стиле всякий раз, когда отображается кадр стека (что включает каждую остановку вашей программы). Этот распознаваемый формат выглядит как два знака '`\032`', за которыми следует имя файла, номер строки

и символьная позиция, разделенные двоеточиями, и знак новой строки. Программа интерфейса Emacs-GDB использует два знака '`\032`' как сигнал для отображения исходного текста для кадра.

`-epoch` Интерфейс Emacs-GDB Epoch устанавливает этот ключ, когда вызывает GDB как подпроцесс. Это велит GDB изменить свои подпрограммы печати так, чтобы позволить Epoch отображать значения выражений в отдельном окне.

`-annotate` *уровень*
Этот ключ устанавливает *уровень примечаний* внутри GDB. Его эффект аналогичен использованию '`set annotate` *уровень*' (см. Глава 18 [Примечания], с. 163). Уровень примечаний контролирует, какое количество информации GDB выводит вместе с приглашением, значениями выражений, строками исходного текста и другими типами вывода. Уровень 0 является обычным, уровень 1 используется, когда GDB выполняется как подпроцесс GNU Emacs, уровень 2 выводит максимальное количество примечаний и подходит для программ, которые управляют GDB.

`-async` Использовать асинхронный цикл событий для интерфейса командной строки. GDB обрабатывает все события, такие как ввод пользователя с клавиатуры, через специальный цикл событий. Это позволяет GDB принимать и обрабатывать команды пользователя параллельно с выполнением отлаживаемого процесса[1], так что вы не должны ждать возвращения управления GDB, прежде чем ввести следующую команду. (*Замечание:* в версии 5.0, асинхронное выполнение на целевой системе еще не поддерживается, так что режим '`-async`' еще реализован не полностью.)

Когда стандартный ввод соединен с терминальным устройством, GDB по умолчанию использует асинхронный цикл событий, если это не отключено ключем '`-noasync`'.

`-noasync` Отключить асинхронный цикл событий для интерфейса командной строки.

`-baud` *бод-в-сек*
`-b` *бод-в-сек*
Устанавливает скорость линии (скорость в бодах, или в битах в секунду) любого последовательного интерфейса, используемого GDB для удаленной отладки.

`-tty` *устройство*
`-t` *устройство*
Запуститься, используя *устройство* для стандартного ввода и вывода вашей программы.

`-interpreter` *интерп*
Использовать интерпретатор *интерп* к качестве интерфейса с управляющей программой или устройством. Подразумевается, что этот ключ должет устанавливаться программами, которые взаимодействуют с GDB, используя его как выходной буфер. Например, '`-interpreter=mi`' велит GDB использовать *интерфейс gdbmi* (см. Глава 19 [Интерфейс GDB/MI], с. 171).

`-write` Открыть выполняемый файл и файл дампа памяти как для чтения, так и для записи. Это эквивалентно команде GDB '`set write on`' (см. Раздел 11.6 [Внесение изменений], с. 104).

`-statistics`
Этот ключ велит GDB печатать статистику о времени и использовании памяти после завершения каждой команды и возврата к приглашению.

[1] GDB, собранный со средствами DJGPP для MS-DOS/MS-Windows, поддерживает этот режим функционирования, но цикл событий приостанавливается, когда выполняется отлаживаемая программа.

`-version` Этот ключ велит GDB напечатать номер своей версии и объявление об отсутствии гарантий, и затем завершиться.

2.2 Выход из GDB

`quit` [*выражение*]
`q` Чтобы выйти из GDB, используйте команду `quit` (сокращенно `q`), или введите знак конца файла (обычно `C-d`). Если вы не укажете *выражение*, GDB закончит работу нормально; в противном случае, он использует результат *выражения* как код ошибки.

Прерывание (часто `C-c`) не приводит к выходу из GDB, а завершает любую выполняющуюся команду и возвращает вас на командный уровень. Вы можете безопасно пользоваться прерыванием в любое время, потому что GDB не позволяет ему вступить в силу до того, как это станет безопасным.

Если вы использовали GDB для управления присоединенным процессом или устройством, вы можете освободить его командой `detach` (см. Раздел 4.7 [Отладка запущенного ранее процесса], с. 25).

2.3 Команды оболочки

Если вам случайно потребовалось выполнить команды оболочки в течение сеанса отладки, нет смысла приостанавливать или покидать GDB; вам достаточно воспользоваться командой `shell`.

`shell` *командная строка*
Вызвать стандартную оболочку для выполнения *командной строки*. Переменная окружения `SHELL`, если она существует, определяет, какую оболочку запустить. В противном случае, GDB использует оболочку по умолчанию (‘`/bin/sh`’ в системах Unix, ‘`COMMAND.COM`’ в MS-DOS, и так далее).

В средах разработки часто бывает необходимо воспользоваться утилитой `make`. Для этой цели вам не обязательно пользоваться командой `shell` в GDB:

`make` *make-apr*
Выполнить программу `make` с указанными аргументами. Это эквивалентно ‘`shell make` *make-apr*’.

3 Команды GDB

Вы можете сокращать команды GDB по нескольким первым символам имени команды, если это сокращение однозначно; и вы можете повторять определенные команды GDB простым нажатием ⟨RET⟩. Вы также можете использовать клавишу ⟨TAB⟩ для того, чтобы GDB сам дополнил остаток слова в команде (или показал вам возможные альтернативы, если существует несколько вариантов).

3.1 Синтаксис команд

Команда GDB представляет собой одну строку ввода. Никаких ограничений на ее длину нет. Она начинается именем команды, за которым следуют параметры, значение которых определяется ее названием. Например, команда `step` допускает в качестве параметра число шагов, как в '`step 5`'. Вы также можете использовать команду `step` и без параметров. Некоторые команды не допускают никаких параметров.

Названия команд GDB всегда могут быть сокращены, если это сокращение однозначно. Другие возможные сокращения команд перечислены в документации по отдельным командам. В некоторых случаях, допускаются даже неоднозначные сокращения; напрмер, `s` специально определено как эквивалент `step`, даже если существуют другие команды, чьи названия начинаются на `s`. Вы можете проверить сокращения, задавая их как параметр для команды `help`.

Введенная пустая строка (просто нажатие ⟨RET⟩), означает повтор предыдущей команды. Определенные команды (например, `run`) не повторяются таким способом; это те команды, непреднамеренное повторение которых может вызвать проблемы и которые вы вряд ли захотите повторять.

Команды `list` и `x`, при их повторе нажатием ⟨RET⟩, вместо точного повтора создают новые параметры. Это позволяет легко просматривать исходный текст или память.

GDB может также использовать ⟨RET⟩ по-другому: для разделения длинного вывода, аналогично обычной утилите `more` (см. Раздел 15.4 [Размер экрана], с. 153). Так как в такой ситуации легко нажать ⟨RET⟩ слишком много раз, GDB блокирует повтор после любой команды, генерирующей такой тип вывода.

Любой текст, расположенный от `#` до конца строки является комментарием; он ничего не делает. В основном, это полезно в командных файлах (см. Раздел 16.3 [Командные файлы], с. 158).

3.2 Завершение команд

GDB может дополнить за вас окончание слова в команде, если существует только один вариант; он также может в любой момент показать, какие завершения возможны для следующего слова в команде. Это работает для команд и подкоманд GDB, а также для имен символов в вашей программе.

Нажмите ⟨TAB⟩ в любой момент, когда захотите, чтобы GDB дополнил оставшуюся часть слова. Если существует только одна возможность, GDB дополняет слово и ждет, пока вы закончите команду (или нажмете ⟨RET⟩, чтобы ввести ее). Например, если вы введете

```
(gdb) info bre ⟨TAB⟩
```

GDB дополнит остаток слова '`breakpoints`', так как у команды `info` есть единственная подкоманда, начинающаяся с '`bre`':

```
(gdb) info breakpoints
```

Теперь вы можете нажать либо ⟨RET⟩, чтобы выполнить команду `info breakpoints`, либо удалить часть символов и ввести что-то другое, если ‘`breakpoints`’ не является той командой, которую вы ожидали. (Если вы с самого начала были уверены, что хотите `info breakpoints`, вы также могли просто нажать ⟨RET⟩, сразу после ‘`info bre`’, использовав сокращение команды вместо завершения).

Если при нажатии ⟨TAB⟩ существует несколько вариантов для следующего слова, GDB издает звук. Вы можете либо ввести больше символов и попробовать снова, либо нажать ⟨TAB⟩ второй раз; GDB выведет все возможные завершения для этого слова. Например, вы можете захотеть установить точку останова на подпрограмме, чье имя начинается с ‘`make_`’, но когда вы вводите *`b make_`*⟨TAB⟩, GDB лишь издает звук. Повторное нажатие ⟨TAB⟩ отображает все имена функций в вашей программе, начинающиеся с этих символов. Например:

```
(gdb) b make_ ⟨TAB⟩
```

GDB издает звук; нажав ⟨TAB⟩ еще раз, видим:

```
make_a_section_from_file     make_environ
make_abs_section             make_function_type
make_blockvector             make_pointer_type
make_cleanup                 make_reference_type
make_command                 make_symbol_completion_list
(gdb) b make_
```

После отображения возможных вариантов, GDB копирует ваш частичный ввод (‘`b make_`’ в этом примере), так что вы можете закончить команду.

Если вы сначала хотите только увидеть список альтернатив, вы можете нажать *`M-?`*, а не нажимать ⟨TAB⟩ дважды. *`M-?`* означает ⟨META⟩ *`?`*. Вы можете ввести это либо удерживая клавишу, обозначенную на вашей клавиатуре как ⟨META⟩ (если такая есть), и нажать *`?`*, или как ⟨ESC⟩, за которой следует *`?`*.

Иногда нужная вам строка, являясь логически “словом”, может содержать скобки или другие символы, которые GDB обычно исключает из своего понятия о слове. Чтобы позволить завершению слов работать в такой ситуации, вы можете заключить слова в командах GDB в `’` (знаки одинарных кавычек).

Наиболее вероятная ситуация, где вам это может потребоваться, это при вводе имени функции в Си++. Это происходит потому, что Си++ допускает перегрузку функций (множественные определения одной и той же функции, различающиеся типом параметров). Например, когда вы хотите установить точку останова, вам может потребоваться различать, имеете вы в виду версию `name` с параметром целого типа, `name(int)`, или же версию с параметром вещественного типа, `name(float)`. Для использования возможностей завершения слов в такой ситуации, введите одинарную кавычку `’` в начале имени функции. Это предупреждает GDB, что ему может потребоваться принять во внимание больше информации, чем обычно, когда вы нажимаете ⟨TAB⟩ или *`M-?`* для запроса завершения слова:

```
(gdb) b ’bubble( M-?
bubble(double,double)    bubble(int,int)
(gdb) b ’bubble(
```

В некоторых случаях, GDB может сам определить, что завершение имени требует использования кавычек. Когда это происходит, GDB вставляет кавычку за вас (выполняя завершение на столько, на сколько это возможно), если вы не ввели ее в первой позиции:

```
(gdb) b bub ⟨TAB⟩
```

GDB изменяет вашу строку ввода на следующую, и издает звук:

```
(gdb) b ’bubble(
```

Вообще, GDB может определить, что кавычка нужна (и вставляет ее), если вы запрашиваете завершение перегруженного символа до того, как начали вводить список параметров.

Для большей информации о перегруженных функциях, смотрите Раздел 9.4.1.3 [Выражения Си++], с. 84. Вы можете использовать команду `set overload-resolution off` для отключения распознавания перегруженных символов; смотрите Раздел 9.4.1.7 [Возможности GDB для Си++], с. 85.

3.3 Получение справки

Используя команду `help`, вы всегда можете запросить информацию о командах у самого GDB.

`help`
`h`
Вы можете использовать `help` (сокращенно `h`) без параметров, для отображения короткого списка именованных классов команд:

```
(gdb) help
List of classes of commands:

aliases - Aliases of other commands
breakpoints - Making program stop at certain points
data - Examining data
files - Specifying and examining files
internals - Maintenance commands
obscure - Obscure features
running - Running the program
stack - Examining the stack
status - Status inquiries
support - Support facilities
tracepoints - Tracing of program execution without
                stopping the program
user-defined - User-defined commands

Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

`help` *класс*
Используя один из общих классов справки как параметр, вы можете получить список отдельных команд этого класса. Вот, например, отображение справки для класса `status`:

```
(gdb) help status
Status inquiries.

List of commands:

info - Generic command for showing things
 about the program being debugged
show - Generic command for showing things
 about the debugger

Type "help" followed by command name for full
```

```
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

`help` *команда*
: Если указать имя команды в качестве параметра `help`, GDB выведет короткую справку о том, как ей пользоваться.

`apropos` *арг*
: Команда `apropos` *арг* производит поиск по регулярному выражению, заданному в *арг*, во всех командах GDB и их документации. Она выводит все найденные совпадения. Например:

```
apropos reload
```

приводит к:

```
set symbol-reloading - Set dynamic symbol table reloading
                                    multiple times in one run
show symbol-reloading - Show dynamic symbol table reloading
                                    multiple times in one run
```

`complete` *арг*
: Команда `complete` *арг* перечисляет все возможные завершения для начала команды. Используйте *арг* для задания начала команды, которую вы хотите завершить. Например,

```
complete i
```

приводит к:

```
if
ignore
info
inspect
```

Это предназначено для использования GNU Emacs.

В дополнение к `help`, вы можете использовать команды GDB `info` и `show` для получения информации о состоянии вашей программы, или о состоянии самого GDB. Каждая команда поддерживает много тем запросов; это руководство описывает каждую тему в соответствующем месте. Списки в разделах `info` и `show` в Алфавитном указателе указывают на все подкоманды. См. [Алфавитный указатель], с. 253.

`info`
: Эта команда (сокращенно `i`) предназначена для описания состояния вашей программы. Например, вы можете с помощью `info args` просмотреть аргументы, переданные вашей программе, с помощью `info registers` перечислить используемые в настоящий момент регистры, или используя `info breakpoints` вывести установленные вами точки останова. Вы можете получить полный список подкоманд команды `info` с помощью `help info`.

`set`
: Вы можете присвоить переменной среды результат выражения с помощью `set`. Например, вы можете установить приглашение GDB в знак $ используя `set prompt $`.

`show`
: В отличие от `info`, команда `show` предназначена для описания состояния самого GDB. Вы можете изменить почти все, что показывает `show`, используя соответствующую команду `set`. Например, командой `set radix` вы можете задать, какую систему счисления использовать для вывода, или просто узнать, какая система используется в данный момент с помощью команды `show radix`.

  Для отображения всех устанавливаемых параметров и их текущих значений, вы можете использовать `show` без аргументов; также можно использовать `info set`. Обе команды приводят к одинаковому результату.

Вот три разнообразные подкоманды `show`, которые не имеют соответствующих `set`-команд:

`show version`
: Показывает, какая версия GDB запущена. Вам следует включать эту информацию в отчеты об ошибках в GDB. Если вы используете несколько версий GDB, вам может потребоваться определить, какая из них запущена; по мере развития отладчика появляются новые команды, а старые могут исчезнуть. Кроме того, многие распространители операционных систем поставляют модифицированные версии GDB, также существуют модифицированные версии GDB в GNU/Linux. Номер версии—это номер, появляющийся при старте.

`show copying`
: Выводит информацию о правах на распространение GDB.

`show warranty`
: Отображает заявление GNU об отсутствии гарантий, или гарантии, если ваша версия GDB поставляется с гарантиями.

4 Выполнение программ под управлением GDB

Прежде чем выполнять программу под управлением GDB, при компиляции вы должны сгенерировать отладочную информацию.

Вы можете запустить GDB с параметрами или без, в любой среде по вашему выбору. Если вы отлаживаете программу на той же машине, на которой выполняется GDB, вы можете перенаправить ввод и вывод вашей программы, отлаживать уже выполняющийся процесс, или убить дочерний процесс.

4.1 Компиляция для отладки

Для эффективной отладки программы, при компиляции вы должны сгенерировать отладочную информацию. Эта отладочная информация сохраняется в объектном файле; она описывает тип данных каждой переменной или функции, и соответствие между номерами строк исходного текста и адресами в выполняемом коде.

Чтобы запросить генерацию отладочной информации, укажите ключ '`-g`' при вызове компилятора.

Многие компиляторы Си не могут обрабатывать ключи '`-g`' и '`-O`' вместе. Используя такие компиляторы, вы не можете создавать оптимизированные выполняемые файлы, содержащие отладочную информацию.

GCC, GNU компилятор Си, поддерживает '`-g`' с или без '`-O`', делая возможным отладку оптимизированного кода. Мы рекомендуем, чтобы вы *всегда* использовали '`-g`' при компиляции программ. Вы можете думать, что ваша программа правильная, но нет никакого смысла испытывать удачу.

Когда вы отлаживаете программу, откомпилированную с '`-g -O`', помните, что оптимизатор перестраивает ваш код; отладчик же показывает то, что там находится в действительности. Не удивляйтесь, если порядок выполнения не будет в точности соответствовать вашему исходному файлу! Крайний пример: если вы определяете переменную, но нигде ее не используете, GDB никогда не увидит этой переменной, потому что при оптимизации компилятор ее исключит.

Некоторые вещи не работают с '`-g -O`' также, как просто с '`-g`', в частности, на машинах с планированием инструкций. Если сомневаетесь, перекомпилируйте с одним ключем '`-g`', и если это устранит проблему, пожалуйста, сообщите нам об этом как об ошибке (включите тестовый пример!).

Ранние версии компилятора GNU Си допускали вариант ключа для отладочной информации '`-gg`'. GDB больше не поддерживает этот формат; если этот ключ есть у вашего компилятора GNU Си, не используйте его.

4.2 Начало выполнения вашей программы

`run`
`r`
: Используйте команду `run` для запуска вашей программы под управлением GDB. Сначала вы должны задать имя программы (кроме как на VxWorks) с параметрами GDB (см. Глава 2 [Вход и выход из GDB], с. 9), или используя команды `file` или `exec-file` (см. Раздел 12.1 [Команды для задания файлов], с. 105).

Если вы запускаете вашу программу в среде выполнения, поддерживающей процессы, `run` создает подчиненный процесс, и этот процесс выполняет вашу программу. (В средах, не поддерживающих процессы, `run` выполняет переход на начало вашей программы.)

Выполнение программы зависит от определенной информации, которую она получает от породившего ее процесса. GDB предоставляет способы задать эту информацию, что вы должны сделать *до* запуска программы. (Вы можете изменить ее после старта, но такие изменения воздействуют на вашу программу только при следующем запуске.) Эта информация может быть разделена на четыре категории:

Параметры.
: Задайте параметры, которые нужно передать вашей программе, как параметры команды `run`. Если на вашей системе доступна оболочка, она используется для передачи параметров, так что при их описании вы можете использовать обычные соглашения (такие как раскрывание шаблонов или подстановка переменных). В системах Unix, вы можете контролировать, какая оболочка используется, с помощью переменной среды `SHELL`. См. Раздел 4.3 [Аргументы вашей программы], с. 22.

Среда.
: Обычно ваша программа наследует свою среду от GDB, но вы можете использовать команды GDB `set environment` и `unset environment`, чтобы изменить часть настроек среды, влияющих на нее. См. Раздел 4.4 [Рабочая среда вашей программы], с. 23.

Рабочий каталог.
: Ваша программа наследует свой рабочий каталог от GDB. Вы можете установить рабочий каталог GDB командой `cd`. См. Раздел 4.5 [Рабочий каталог вашей программы], с. 24.

Стандартный ввод и вывод.
: Обычно ваша программа использует те же устройства для стандартного ввода и вывода, что и GDB. Вы можете перенаправить ввод и вывод в строке команды `run`, или использовать команду `tty`, чтобы установить другое устройство для вашей программы. См. Раздел 4.6 [Ввод и вывод вашей программы], с. 24.

  Предупреждение: Хотя перенаправление ввода и вывода работает, вы не можете использовать каналы для передачи выходных данных отлаживаемой программы другой программе; если вы попытаетесь это сделать, скорее всего GDB перейдет к отладке неправильной программы.

Когда вы подаете команду `run`, ваша программа начинает выполняться немедленно. См. Глава 5 [Остановка и продолжение], с. 31, для обсуждения того, как остановить вашу программу. Как только ваша программа остановилась, вы можете вызывать функции вашей программы, используя команды `print` или `call`. См. Глава 8 [Исследование данных], с. 61.

Если время модификации вашего символьного файла изменилось с того момента, когда GDB последний раз считывал символы, он уничтожает свою символьную таблицу и считывает ее заново. При этом GDB старается сохранить ваши текущие точки останова.

4.3 Аргументы вашей программы

Аргументы к вашей программе могут быть заданы как аргументы к команде `run`. Они передаются оболочке, которая раскрывает символы шаблонов и выполняет перенаправление ввода-вывода, и с того момента попадают в вашу программу. Ваша переменная среды `SHELL` (если она существует) определяет, какую оболочку использует GDB. Если вы не определите `SHELL`, он использует оболочку по умолчанию (‘`/bin/sh`’ в Unix).

В не-Unix системах, программу обычно запускает непосредственно GDB, который эмулирует перенаправление ввода-вывода через соответствующие системные вызовы, и символы шаблонов раскрываются кодом запуска, а не оболочкой.

`run` без аргументов использует те же аргументы, которые использовались предыдущей командой `run`, или которые установлены командой `set args`.

`set args` Задает аргументы, которые будут использоваться при следующем запуске вашей программы. Если у `set args` нет аргументов, `run` выполняет вашу программу без аргументов. Если вы запустили вашу программу с аргументами, то единственный способ запустить ее снова без аргументов—это использовать `set args` до следующего запуска командой `run`.

`show args` Показать аргументы, которые будут переданы вашей программе при ее вызове.

4.4 Рабочая среда вашей программы

Среда состоит из набора переменных среды и их значений. Переменные среды обычно хранят такие данные, как ваше имя пользователя, домашний каталог, тип терминала и путь поиска для запуска программ. Как правило, вы устанавливаете переменные среды с помощью оболочки и они наследуются всеми другими программами, которые вы вызываете. При отладке может оказаться полезным попробовать запустить программу в измененной среде, не перезапуская GDB.

`path` *каталог*

Добавить *каталог* в начало переменной среды `PATH` (пути поиска выполняемых файлов), как для GDB, так и для вашей программы. Вы можете указать названия нескольких каталогов, разделив их пробелом или системно-зависимым разделителем (‘`:`’ в Unix, ‘`;`’ в MS-DOS и MS-Windows). Если *каталог* уже находится в списке путей, он переносится в начало, так что поиск в нем будет производиться раньше.

Вы можете использовать строку ‘`cwd`’, чтобы сослаться на рабочий каталог, который является текущим в тот момент, когда GDB производит поиск. Если вместо этого вы используете ‘`.`’, то она будет указывать на тот каталог, в котором вы выполнили команду `path`. GDB заменяет ‘`.`’ в аргументе *каталог* (на текущий путь) до добавления *каталога* к списку путей поиска.

`show paths`

Отобразить список путей для поиска выполняемых файлов (переменную среды `PATH`).

`show environment` [*имя-перем*]

Вывести значение переменной среды *имя-перем*, которое будет передано вашей программе при ее старте. Если вы не указываете *имя-перем*, вывести названия и значения всех переменных среды, передаваемых вашей программе. Вы можете сократить `environment` как `env`.

`set environment` *имя-перем* [`=`*значение*]

Присваивает *значение* переменной среды *имя-перем*. Значение меняется только для вашей программы, но не для самого GDB. *значение* может быть любой строкой; значениями переменных среды являются просто строки, а их интерпретацию обеспечивает ваша программа. Параметр *значение* является необязательным; если он опущен, переменная устанавливается в пустое значение.

Например, эта команда:

```
set env USER = foo
```

говорит отлаживаемой программе, что при последующих запусках именем пользователя является ‘`foo`’. (Пробелы, окружающие ‘`=`’, использованы здесь для ясности; в действительности, они не обязательны.)

`unset environment` *имя-перем*

Удалить переменную *имя-перем* из среды, передаваемой вашей программе. Это отличается от ‘`set env` *имя-перем* `=`’; `unset environment` удаляет переменную из среды, а не присваивает ей пустое значение.

Предупреждение: В системах Unix, GDB вызывает вашу программу, используя оболочку, указанную вашей переменной среды `SHELL`, если она определена (или `/bin/sh`, если не определена). Если ваша переменная `SHELL` указывает на оболочку, которая выполняет файл инициализации—такой как '`.cshrc`' для оболочки C-shell, или '`.bashrc`' для BASH—любая переменная, которую вы установите в этом файле, воздействует на вашу программу. В этой связи, вы можете захотеть перенести установку переменных среды в файлы, которые выполняются только при вашем входе в систему, такие как '`.login`' или '`.profile`'.

4.5 Рабочий каталог вашей программы

Каждый раз, когда вы запускаете свою программу командой `run`, она наследует рабочий каталог от текущего рабочего каталога GDB. В начальный момент, рабочий каталог GDB наследуется от его родительского процесса (обычно оболочки), но вы можете задать новый рабочий каталог командой `cd` из GDB.

Рабочий каталог GDB также служит каталогом по умолчанию для команд отладчика, определяющих действия с файлами. См. Раздел 12.1 [Команды для задания файлов], с. 105.

`cd` *каталог*
: Установить рабочий каталог GDB в *каталог*.

`pwd`
: Вывести рабочий каталог GDB.

4.6 Ввод и вывод вашей программы

По умолчанию, программа, которую вы запускаете под управлением GDB, осуществляет ввод и вывод на тот же терминал, что и GDB. Для взаимодействия с вами, GDB переключает терминал в свой собственный терминальный режим, но он записывает терминальные режимы, которые использовала ваша программа, и переключается назад к ним, когда вы продолжаете выполнение программы.

`info terminal`
: Отображает информацию, записанную GDB о терминальных режимах, которые использует ваша программа.

Вы можете перенаправить ввод и/или вывод вашей программы, используя перенаправление оболочки с помощью команды `run`. Например,

```
run > выходной-файл
```

запускает вашу программу, перенаправляя ее вывод в '`выходной-файл`'.

Другой способ задать, как ваша программа должна осуществлять ввод и вывод—использовать команду `tty`. Эта команда принимает в качестве агрумента имя файла, который будет использоваться по умолчанию для будущих команд `run`. Эта команда также сбрасывает управляющий терминал для дочернего процесса для будущих команд `run`. Например,

```
tty /dev/ttyb
```

указывает, что процессы, запущенные последующими командами `run`, для ввода и вывода используют по умолчанию терминал '`/dev/ttyb`', и что он будет их управляющим терминалом.

Явное перенаправление в `run` замещает эффект команды `tty` для устройств ввода-вывода, но не ее воздействие на управляющий терминал.

Когда вы используете команду `tty` или перенаправляете ввод в команде `run`, изменяется только ввод *для вашей программы*. Ввод для GDB по прежнему происходит через ваш терминал.

4.7 Отладка запущенного ранее процесса

`attach` *идент-процесса*

Эта команда присоединяется к выполняющемуся процессу—процессу, который был запущен вне GDB. (Команда `info files` показывает ваши активные цели.) В качестве аргумента команда получает идентификатор процесса. Обычный способ узнать идентификатор Unix-процесса—воспользоваться утилитой `ps` или командой оболочки '`jobs -l`'.

`attach` не повторяется, если вы нажмете ⟨RET⟩ второй раз после выполнения команды.

Чтобы использовать `attach`, ваша программа должна выполняться в среде, поддерживающей процессы; например, `attach` не работает на специальных машинах, не имеющих операционной системы. Вы также должны обладать полномочиями для посылки сигнала процессу.

Когда вы используете `attach`, отладчик находит программу выполняющегося процесса, производя поиск сперва в текущем рабочем каталоге, а затем (если программа не найдена), используя пути поиска исходных файлов (см. Раздел 7.3 [Определение каталогов с исходными файлами], с. 57). Также, для загрузки программы вы можете использовать команду `file`. См. Раздел 12.1 [Команды для задания файлов], с. 105.

Первое, что GDB делает после подготовки указанного процесса к отладке—останавливает его. Вы можете исследовать и изменять присоединенный процесс всеми командами GDB, которые обычно доступны, когда вы запускаете процессы с помощью `run`. Вы можете устанавливать точки останова; вы можете пошагово выполнять программу и продолжить ее обычное выполнение, вы можете изменять области данных. Если вы решите продолжить выполнение процесса после присоединения к нему GDB, вы можете использовать команду `continue`.

`detach` Когда вы закончили отлаживать присоединенный процесс, для его освобождения из под управления GDB вы можете использовать команду `detach`. Отсоединение процесса продолжает его выполнение. После команды `detach`, этот процесс и GDB снова становятся совершенно независимыми, и вы готовы присоединить или запустить с помощью `run` другой процесс. `detach` не повторяется, если вы нажмете ⟨RET⟩ еще раз после выполнения команды.

Если вы выйдете из GDB или используете команду `run`, пока у вас есть присоединенный процесс, вы убьете этот процесс. По умолчанию, GDB запрашивает подтверждение, если вы пытаетесь сделать одну из этих вещей; вы можете контролировать, нужно вам это подтверждение или нет, используя команду `set confirm` (см. Раздел 15.6 [Необязательные предупреждения и сообщения], с. 154).

4.8 Уничтожение дочернего процесса

`kill` Уничтожить дочерний процесс, в котором ваша программа выполняется под управлением GDB.

Эта команда полезна, если вы хотите отладить дамп памяти, а не выполняющийся процесс. GDB игнорирует любые дампы памяти, пока ваша программа выполняется.

В некоторых операционных системах, программа не может быть выполнена вне GDB, пока у вас есть в ней точки останова, установленные отладчиком. В этой ситуации вы можете использовать команду `kill`, чтобы разрешить выполнение вашей программы вне отладчика.

Команда `kill` также полезна, если вы хотите перекомпилировать и перекомпоновать вашу программу, так как во многих системах невозможно модифицировать исполняемый файл во время выполнения процесса. В этом случае, когда вы в следующий раз введете `run`, GDB заметит, что файл изменился, и заново прочитает символьную таблицу (стараясь при этом сохранить ваши точки останова).

4.9 Отладка программ с несколькими нитями

В некоторых операционных системах, таких как HP-UX и Solaris, одна программа может иметь несколько *нитей* выполнения. Точная семантика нитей меняется от одной операционной системы к другой, но в общем, нити одной программы сродни нескольким процессам—за исключением того, что они разделяют одно адресное пространство (то есть, все они могут исследовать и модифицировать одни и те же переменные). С другой стороны, каждая нить имеет свои собственные регистры и стек выполнения, и, возможно, свои собственные участки памяти.

GDB предоставляет следующие возможности для отладки многонитевых программ:

- автоматическое уведомление о новых нитях
- '`thread` *номер-нити*', команда для переключения между нитями
- '`info threads`', команда для запроса информации о существующих нитях
- '`thread apply` [*номер-нити*] [*all*] *арг*', команда для применения некоторой команды к списку нитей
- точки останова, определяемые отдельно для каждой нити

Предупреждение: Эти возможности доступны еще не в любой конфигурации GDB, где операционная система поддерживает нити. Если ваш GDB не поддерживает нити, эти команды не имеют эффекта. Например, в системах без поддержки нитей, GDB ничего не выводит на команду '`info threads`', и всегда отвергает команду `thread`, как в этом примере:

```
(gdb) info threads
(gdb) thread 1
Thread ID 1 not known.  Use the "info threads" command to
see the IDs of currently known threads.[1]
```

Возможности отладки нитей GDB позволяют вам наблюдать все нити во время выполнения вашей программы, но когда управление переходит к GDB, одна конкретная нить выделяется для отладки. Она называется *текущей нитью*. Отладочные команды показывают информацию о программе с точки зрения текущей нити.

Когда GDB обнаруживает новую нить в вашей программе, он выводит для нее идентификатор на целевой системе с сообщением в форме '`[New` *сист-тег*`]`'. *Сист-тег* является идентификатором нити, чья форма различается в зависимости от конкретной системы. Например, в LynxOS вы можете увидеть

```
[New process 35 thread 27]
```

когда GDB замечает новую нить. Напротив, в системе SGI, *сист-тег* выглядит просто как '`process 368`', без дополнительных спецификаций.

Для отладочных целей, GDB присваивает свои собственные номера нитей—всегда в виде одного целого числа—каждой нити в вашей программе.

`info threads`

Вывести краткую информацию обо всех имеющихся в данный момент в вашей программе нитях. Для каждой нити, GDB отображает (в этом порядке):

[1] Нить с идентификатором 1 неизвестна. Используйте команду "info threads", чтобы получить идентификаторы известных нитей. *(Прим. переводчика)*

1. номер нити, назначенный GDB
2. идентификатор нити на целевой системе (*сист-тег*)
3. краткие сведения о текущем кадре стека для этой нити

Звездочка '`*`' слева от номера нити GDB обозначает текущую нить.

Например,

```
(gdb) info threads
  3 process 35 thread 27  0x34e5 in sigpause ()
  2 process 35 thread 23  0x34e5 in sigpause ()
* 1 process 35 thread 13  main (argc=1, argv=0x7ffffff8)
    at threadtest.c:68
```

В системах HP-UX:

Для отладочных целей, GDB присваивает свои собственные номера нитей—небольшие целые, присваиваемые в порядке создания нитей— каждой нити в вашей программе.

Когда GDB обнаруживает новую нить в вашей программе, он выводит как номер нити, присвоенный GDB, так и идентификатор на целевой системе для нити с сообщением в форме '`[New` *сист-тег*`]`'. *сист-тег* является идентификатором нити, чья форма различается в зависимости от конкретной системы. Например, в HP-UX, когда GDB замечает новую нить, вы увидите

```
[New thread 2 (system thread 26594)]
```

`info threads`

Вывести краткую информацию обо всех имеющихся в данный момент в вашей программе нитях. Для каждой нити, GDB отображает (в этом порядке):

1. номер нити, назначенный GDB
2. идентификатор нити на целевой системе (*сист-тег*)
3. краткие сведения о текущем кадре стека для этой нити

Звездочка '`*`' слева от номера нити GDB означает текущую нить.

Например,

```
(gdb) info threads
    * 3 system thread 26607  worker (wptr=0x7b09c318 "@") \

                               at quicksort.c:137
      2 system thread 26606  0x7b0030d8 in __ksleep () \

                               from /usr/lib/libc.2
      1 system thread 27905  0x7b003498 in _brk () \

                               from /usr/lib/libc.2
```

`thread` *номер-нити*

Сделать нить с номером *номер-нити* текущей. Аргумент команды, *номер-нити*, является внутренним номером нити GDB, который показан в первом поле '`info threads`'. GDB отвечает, выводя системный идентификатор выбранной вами нити, и обзор ее кадра стека:

```
(gdb) thread 2
[Switching to process 35 thread 23]
0x34e5 in sigpause ()
```

Также как и с сообщением '`[New ...]`', форма текста после '`Switching to`' зависит от соглашений для идентификации нитей в вашей системе.

`threads apply` [*номер-нити*] [*all*] *арг*

Команда `thread apply` позволяет вам применить команду к одной или нескольким нитям. Задайте номера нитей, на которые вы хотите воздействовать, в аргументе *номер-нити*. *Номер-нити*—это внутренний номер нити GDB, который показан в первом поле '`info threads`'. Чтобы применить команду ко всем нитям, используйте `thread apply all` *арг*.

Когда GDB останавливает вашу программу, вследствие точки останова или по сигналу, он автоматически выбирает нить, в которой появилась точка останова или сигнал. GDB предупреждает вас о переключении контекста сообщением в форме '`[Switching to` *систтег*`]`' для идентификации нити.

См. Раздел 5.4 [Остановка и запуск многонитевых программ], с. 48, для дополнительной информации о поведении GDB, когда вы останавливаете и запускаете многонитевую программу.

См. Раздел 5.1.2 [Установка точек наблюдения], с. 35, для информации о точках наблюдения в многонитевых программах.

4.10 Отладка многонитевых программ

В большинстве систем, GDB не имеет специальной поддержки для отладки программ, создающих дополнительные процессы с помощью функции `fork`. Когда программа вызывает `fork`, GDB будет продолжать отладку родительского процесса, а дочерний процесс будет выполняться беспрепятственно. Если выполнение дочернего процесса дойдет до места, где вы установили точку останова, дочерний процесс получит сигнал `SIGTRAP`, который приведет к остановке процесса (если он не перехватывает этот сигнал).

Однако, если вы хотите отладить дочерний процесс, существует достаточно простое решение. Поместите вызов `sleep` в код программы, который дочерний процесс выполнит после `fork`. Может быть удобным вызывать `sleep` только если установлена определенная переменная среды или если существует определенный файл, так что задержка не будет происходить, если вы не захотите отлаживать дочерний процесс. Пока дочерний процесс спит, используйте программу `ps` для получения ее идентификатора процесса. Затем укажите GDB (новому экземпляру GDB, если вы отлаживаете также и родительский процесс) присоединиться к дочернему процессу (см. Раздел 4.7 [Присоединение], с. 25). Начиная с этого момента, вы можете отлаживать дочерний процесс точно также, как любой другой процесс, к которому вы присоединились.

В системе HP-UX (только в версиях 11.x и более поздних?) GDB предоставляет средства для отладки программ, которые создают дополнительные процессы, используя функции `fork` или `vfork`.

По умолчанию, когда программа ветвится, GDB будет продолжать отладку родительского процесса, а дочерний процесс будет выполняться беспрепятственно.

Если вы хотите отлаживать дочерний процесс вместо родительского, используйте команду `set follow-fork-mode`.

`set follow-fork-mode` *режим*

Устанавливает реакцию отладчика на вызов `fork` или `vfork` в программе. Вызов `fork` или `vfork` создает новый процесс. *режим* может быть:

`parent` После ветвления отлаживается исходный процесс. Дочерний процесс выполняется беспрепятственно. Это поведение по умолчанию.

`child` После ветвления отлаживается новый процесс. Родительский процесс выполняется беспрепятственно.

`ask` Отладчик будет запрашивать один из этих вариантов.

`show follow-fork-mode`
: Отображает текущую реакцию отладчика на вызов `fork` или `vfork`.

Если вы запрашиваете отладку дочернего процесса и за `vfork` следует `exec`, GDB выполняет новую программу до первой точки останова, установленной в ней. Если у вас была установлена точка останова на функции `main` в вашей исходной программе, она будет также установлена на `main` в дочернем процессе.

Когда дочерний процесс порождается вызовом `vfork`, вы не можете отлаживать дочерний или родительский процесс до тех пор, пока не завершится вызов `exec`.

Если вы даете GDB команду `run` после выполнения `exec`, новая программа стартует заново. Чтобы перезапустить родительский процесс, используйте команду `file` с именем выполняемого файла родительской программы в качестве аргумента.

Вы можете использовать команду `catch`, чтобы остановить GDB, когда сделан вызов `fork`, `vfork` или `exec`. См. Раздел 5.1.3 [Установка точек перехвата], с. 37.

5 Остановка и продолжение исполнения

Основные цели применения отладчика—остановка вашей программы до ее завершения, или чтобы в случае нарушений в работе вашей программы вы могли выяснить их причину.

Внутри GDB ваша программа может остановиться по нескольким причинам, таким как сигнал, точка останова, или достижение новой строки после команды GDB, такой как `step`. Затем вы можете исследовать и изменять значения переменных, устанавливать новые точки останова и удалять старые, и затем продолжить выполнение. Обычно, выводимые GDB сообщения предоставляют достаточную информацию о состоянии вашей программы, но вы также можете запросить эту информацию явно в любое время.

`info program`
: Отобразить информацию о состоянии вашей программы: выполняется она или нет, каким процессом она является и почему остановлена.

5.1 Точки останова, точки наблюдения и точки перехвата

Точка останова останавливает вашу программу всякий раз, когда ее выполнение достигает определенной точки. Для каждой точки останова вы можете добавлять условия для лучшего управления условиями остановки. Вы можете устанавливать точки останова командой `break` и ее вариантами (см. Раздел 5.1.1 [Установка точек останова], с. 32), чтобы задать место, где должна остановиться ваша программа, по номеру строки, имени функции или точному адресу.

В конфигурациях HP-UX, SunOS 4.x, SVR4 и Alpha OSF/1, вы можете устанавливать точки останова в разделяемых библиотеках до запуска выполняемого файла. В системах HP-UX существует небольшое ограничение: вы должны подождать, пока программа не перестанет выполняться, для установки точек останова в подпрограммах из разделяемой библиотеки, которые не вызываются напрямую из программы (например, подпрограммах, являющихся аргументами вызова `pthread_create`).

Точка наблюдения—это специальная точка останова, которая останавливает вашу программу при изменении значения выражения. Вы должны использовать другую команду для установки точки наблюдения (см. Раздел 5.1.2 [Установка точек наблюдения], с. 35), но помимо этого, вы можете обращаться с ней так же, как с любой другой точкой останова: вы включаете, отключаете и удаляете точки останова и точки наблюдения при помощи одних и тех же команд.

Вы можете установить, что значения из вашей программы должны отображаться автоматически, когда GDB останавливается в точке останова. См. Раздел 8.6 [Автоматическое отображение], с. 66.

Точка перехвата—это другая специализированная точка останова, которая останавливает вашу программу при возникновении события определенного типа, такого как выбрасывание исключения в Си++ или загрузка библиотеки. Также как с точками наблюдения, вы используете другую команду для установки точки перехвата, (см. Раздел 5.1.3 [Установка точек перехвата], с. 37), но помимо этого, вы можете обращаться с ней так же, как с любой другой точкой останова. (Для остановки, когда ваша программа получает сигнал, используйте команду `handle`; смотрите Раздел 5.3 [Сигналы], с. 46.)

Когда вы создаете точку останова, наблюдения или перехвата, GDB присваивает ей номер; эти номера являются последовательными целыми числами, начинающимися с единицы. Во многих командах для управления различными возможностями точек останова, вы используете эти номера для указания, какую точку останова вы хотите изменить. Каждая точка останова может быть *включена* или *отключена*; если точка останова отключена, она не оказывает никакого влияния на вашу программу, пока вы снова не включите ее.

Некоторые команды GDB допускают в качестве указания точек останова, на которые они действуют, их диапазоны. Диапазон точек останова—это или номер одной точки, например ‘5’, или два таких номера, в порядке увеличения, разделенные дефисом, например ‘5-7’. Когда команде задается диапазон точек останова, она действует на все точки останова в этом диапазоне.

5.1.1 Установка точек останова

Точки останова устанавливаются командой `break` (сокращенно `b`). Вспомогательная переменная отладчика ‘`$bpnum`’ хранит номер последней установленной вами точки останова; смотрите Раздел 8.9 [Вспомогательные переменные], с. 73, для обсуждения того, что вы можете делать со вспомогательными переменными.

Вы можете задавать место для установки новой точки останова несколькими способами.

`break` *функция*
: Установить точку останова на входе в функцию *функция*. При использовании языков, допускающих перегрузку символов, таких как Си++, *функция* может ссылаться более чем на одно возможное место останова. См. Раздел 5.1.8 [Меню точки останова], с. 42, для обсуждения такой ситуации.

`break +`*смещение*
`break -`*смещение*
: Установить точку останова через несколько строк впереди или сзади от позиции, на которой выполнение остановилось в текущем выбранном *кадре стека*. (См. Раздел 6.1 [Кадры стека], с. 51, для описания кадров стека.)

`break` *номер-строки*
: Установить точку останова на строке *номер-строки* в текущем исходном файле. Текущий исходный файл—это файл, исходный текст которого отображался последним. Точка останова остановит вашу программу сразу перед выполнением какого-либо кода на этой строке.

`break` *имя-файла*`:`*номер-строки*
: Установить точку останова на строке *номер-строки* в исходном файле *имя-файла*.

`break` *имя-файла*`:`*функция*
: Установить точку останова на входе в *функцию*, находящуюся в файле *имя-файла*. Указание имени файла вместе с именем функции является излишним, за исключением ситуаций, когда несколько файлов содержат одинаково названные функции.

`break *`*адрес*
: Установить точку останова по адресу *адрес*. Вы можете использовать это для установки точек останова в тех частях вашей программы, которые не имеют отладочной информации или исходных файлов.

`break`
: При вызове без аргументов, `break` устанавливает точку останова на инструкции, которая должна быть выполнена следующей в выбранном кадре стека (см. Глава 6 [Исследование стека], с. 51). В любом выбранном кадре, кроме самого внутреннего, это останавливает вашу программу, как только управление возвращается в этот кадр. Это похоже на результат команды `finish` в кадре внутри выбранного кадра—за исключением того, что `finish` не оставляет активной точки останова. Если вы используете `break` без аргументов в самом внутреннем кадре, GDB останавливается, когда в следующий раз достигает текущего места; это может быть полезно внутри циклов.

Обычно GDB игнорирует точки останова, когда он возобновляет выполнение, пока не будет выполнена хотя бы одна инструкция. Если бы он этого не делал, вы не могли бы продолжать выполнение после точки останова, не отключив сперва эту точку останова. Это правило применяется вне зависимости от того, существовала или нет точка останова, когда ваша программа остановилась.

`break ... if` *усл*

Установить точку останова с условием *усл*; каждый раз, когда достигается точка останова, происходит вычисление выражения *усл*, и остановка происходит только если эта величина не равна нулю—то есть, если *усл* истинно. '`...`' означает один из возможных аргументов, перечисленных выше (или отсутствие аргументов), описывающих место остановки. См. Раздел 5.1.6 [Условия остановки], с. 40, для большей информации об условных точках останова.

`tbreak` *арг*

Установить точку останова только до первой активизации. Аргументы *арг* такие же, как для команды `break`, и точка останова устанавливается аналогичным образом, но она автоматически уничтожается после того, как ваша программа первый раз на ней остановится. См. Раздел 5.1.5 [Отключение точек останова], с. 39.

`hbreak` *арг*

Установить аппаратно-поддерживаемую точку останова. Аргументы *арг* такие же, как и для команды `break`, и точка останова устанавливается аналогичным образом, но она требует аппаратной поддержки и некоторые целевые платформы могут ее не иметь. Основной целью этого является отладка кода EPROM/ROM, так что вы можете установить точку останова на инструкции без изменения инструкции. Это может быть использовано с новой генерацией ловушек, предоставляемой SPARClite DSU и некоторыми машинами на базе x86. Эти платформы будут генерировать ловушки, когда программа обращается к некоторым данным или адресу инструкции, которые назначены регистрам отладки. Однако, регистры аппаратных точек останова могут хранить ограниченное число точек останова. Например, на DSU, только две точки останова могут быть установлены одновременно, и GDB будет отвергать эту команду, если используется больше. Удалите или отключите неиспользуемые аппаратные точки останова перед установкой новых (см. Раздел 5.1.5 [Отключение точек останова], с. 39). См. Раздел 5.1.6 [Условия], с. 40.

`thbreak` *арг*

Установить аппаратно-поддерживаемую точку останова, включенную только до первой активизации. Аргументы *арг* такие же, как и для команды `hbreak`, и точка останова устанавливается аналогичным образом. Однако, как в случае команды `tbreak`, точка останова автоматически уничтожается после того, как программа первый раз на ней остановится. Также, как и в случае команды `hbreak`, точка останова требует аппаратной поддержки и некоторые аппаратные платформы могут ее не иметь. См. Раздел 5.1.5 [Отключение точек останова], с. 39. Смотрите также Раздел 5.1.6 [Условия остановки], с. 40.

`rbreak` *рег-выр*

Установить точки останова на всех функциях, удовлетворяющих регулярному выражению *рег-выр*. Эта команда устанавливает безусловные точки останова при всех совпадениях, выводя список всех установленных точек останова. После установки, они рассматриваются точно так же, как точки останова, установленные командой `break`. Вы можете удалять их, отключать, или делать их условными таким же способом, как любые другие точки останова.

Регулярные выражения имеют стандартный синтаксис, используемый такими средствами, как '`grep`'. Заметьте, что это отличается от синтаксиса, исполь-

зуемого оболочками; так, например, `foo*` подходит для всех функций, которые включают `fo`, за которым следует любое число букв `o`. Существует неявное `.*` в начале и в конце введенного вами регулярного выражения, так что для нахождения только тех функций, которые начинаются на `foo`, используйте `^foo`.

При отладке программ, написанных на Си++, `rbreak` полезна для установки точек останова на перегруженных функциях, не являющихся членами никакого специального класса.

`info breakpoints` [*n*]
`info break` [*n*]
`info watchpoints` [*n*]

Вывести таблицу всех установленных и не удаленных точек останова, наблюдения и перехвата, со следующими колонками для каждой точки:

Номер точки останова

Тип Точка останова, наблюдения или перехвата.

План Помечена ли точка останова для отключения или удаления после активации.

Включена или отключена
Включенные точки останова помечаются как '`y`'. '`n`' отмечает отключенные точки.

Адрес Адрес памяти, где расположена точка останова в вашей программе.

Где Файл и номер строки, где расположена точка останова в исходном файле.

Если точка останова условная, `info break` показывает условие на строке, следующей за этой точкой; команды точки останова, если они есть, перечисляются после этого.

`info break` с номером точки останова *n* в качестве аргумента отображает только эту точку. Вспомогательная переменная `$_` и адрес по умолчанию для исследования для команды `x` устанавливаются равными адресу последней из перечисленных точек останова (см. Раздел 8.5 [Исследование памяти], с. 65).

`info break` отображает то число раз, которое точка останова была активирована. Это особенно полезно при использовании вместе с командой `ignore`. Вы можете игнорировать большое число активаций точки останова, посмотреть информацию о точке останова чтобы узнать, сколько раз она активировалась, и затем запустить заново, игнорируя на единицу меньше, чем это число. Это быстро приведет вас к последней активации этой точки останова.

GDB позволяет вам установить любое число точек останова в одном и том же месте вашей программы. В этом нет ничего глупого или бессмысленного. Когда точки останова являются условными, это даже полезно (см. Раздел 5.1.6 [Условия останова], с. 40).

GDB сам иногда устанавливает точки останова в вашей программе для специальных целей, таких как правильная обработка `longjmp` (в программах на Си). Этим внутренним точкам останова присваиваются отрицательные номера, начиная с `-1`; '`info breakpoints`' не отображает их.

Вы можете увидеть эти точки останова с помощью служебной команды GDB '`maint info breakpoints`'.

`maint info breakpoints`

Используя тот же формат, что и '`info breakpoints`', отобразить как точки останова, установленные вами явно, так и те, которые GDB использует для внутренних целей. Внутренние точки останова показываются с отрицательными номерами. Колонка типа определяет, какого типа точка останова показана:

`breakpoint`
: Обычная, явно установленная точка останова.

`watchpoint`
: Обычная, явно установленная точка наблюдения.

`longjmp`
: Внутренняя точка останова, используемая для корректной обработки пошагового выполнения вызовов `longjmp`.

`longjmp resume`
: Внутренняя точка останова на цели `longjmp`.

`until`
: Временная внутренняя точка останова, используемая командой GDB `until`.

`finish`
: Временная внутренняя точка останова, используемая командой GDB `finish`.

`shlib events`
: События в разделяемых библиотеках.

5.1.2 Установка точек наблюдения

Вы можете использовать точку наблюдения для остановки выполнения, как только изменится значение какого-либо выражения, не предсказывая конкретное место, где это может произойти.

В зависимости от вашей системы, точки наблюдения могут быть реализованы программно или аппаратно. GDB осуществляет программную реализацию точек наблюдения путем пошагового выполнения вашей программы и проверки значения переменной на каждом шаге, что в сотни раз медленнее нормального выполнения. (Но тем не менее это может стоить того, найти ошибку в программе, когда вы не представляете, в какой ее части она находится, может быть очень нелегко.)

В некоторых системах, таких как HP-UX, Linux и некоторых других платформах, базирующихся на x86, GDB включает поддержку для аппаратных точек наблюдения, которые не замедляют выполнение вашей программы.

`watch` *выраж*
: Устанавливает точку наблюдения за выражением. GDB остановит программу, когда *выраж* сохраняется программой и его величина изменяется.

`rwatch` *выраж*
: Устанавливает точку наблюдения, которая остановит программу, когда наблюдаемое *выраж* считывается программой.

`awatch` *выраж*
: Устанавливает точку наблюдения, которая остановит программу, когда *выраж* либо считывается, либо сохраняется программой.

`info watchpoints`
: Эта команда печатает список точек наблюдения, останова и перехвата; это то же самое, что и `info break`.

Когда это возможно, GDB устанавливает *аппаратную точку наблюдения*. Аппаратные точки наблюдения выполняются очень быстро, и отладчик сообщает об изменении величины точно в месте инструкции, где это изменение произошло. Если GDB не может установить аппаратную точку наблюдения, он устанавливает программную точку наблюдения, которая выполняется намного медленнее и сообщает об изменении величины на следующем операторе, а не инструкции, после появления изменения.

Когда вы даете команду `watch`, GDB сообщает

`Hardware watchpoint` *номер*`: ` *выраж*

если ему удалось установить аппаратную точку наблюдения.

В настоящее время, команды `awatch` и `rwatch` могут устанавливать только аппаратные точки наблюдения, так как доступы к данным, которые не изменяют величины наблюдаемого выражения, не могут быть замечены без исследования каждой инструкции во время ее выполнения, а GDB пока этого не делает. Если GDB обнаруживает, что не может установить аппаратную точку останова командами `awatch` или `rwatch`, он напечатает сообщение, аналогичное этому:

```
Expression cannot be implemented with read/access
watchpoint.
```
[1]

Иногда GDB не может установить аппаратную точку наблюдения из-за того, что тип данных наблюдаемого выражения занимает больше места, чем допускает аппаратная точка наблюдения на целевой платформе. Например, некоторые системы позволяют наблюдать за областями, занимающими до 4 байт; на таких системах вы не можете устанавливать аппаратные точки наблюдения за выражениями, которые в результате дают число с плавающей точкой двойной точности (которое обычно занимает 8 байт). В качестве одного из решений, можно разбить большую область на несколько меньших областей, и затем наблюдать за каждой из них с помощью отдельной точки наблюдения.

Если вы установите слишком много аппаратных точек наблюдения, GDB может быть не в состоянии задействовать их все, когда вы возобновите выполнение вашей программы. Так как точное количество активных точек наблюдения неизвестно до того момента, когда ваша программа должна возобновить выполнение, GDB может быть не в состоянии предупредить вас об этом, когда вы устанавливаете точку наблюдения, и предупреждение будет напечатано только когда программа возобновит выполнение:

`Hardware watchpoint` *номер*`: Could not insert watchpoint`[2]

Если это происходит, удалите или отключите некоторые точки наблюдения.

SPARClite DSU будет генерировать ловушки, когда программа обращается к некоторым данным или адресу инструкции, которые отведены для отладочных регистров. Для адресов данных, DSU упрощает команду `watch`. Однако, аппаратные регистры точек останова могут принять только две точки наблюдения за данными, и обе точки наблюдения должны быть одного типа. Например, вы можете установить две точки наблюдения с помощью команды `watch`, две с помощью команды `rwatch`, **или** две с помощью команды `awatch`, но вы не можете установить одну точку наблюдения с помощью одной команды, а другую с помощью другой. GDB не примет команду, если вы попытаетесь смешать различные точки наблюдения. Удалите или отключите неиспользуемые точки наблюдения перед установкой новых.

Если вы вызываете функцию интерактивно, используя `print` или `call`, все установленные вами точки наблюдения будут неактивными, до тех пор пока GDB не достигнет точки останова другого типа, или пока вызов не завершится.

GDB автоматически удаляет точки наблюдения, которые наблюдают за локальными переменными, или за выражениями, которые используют такие переменные, когда они выходят из области видимости, то есть когда выполнение покидает блок, в котором эти переменные были определены. В частности, когда отлаживаемая программа завершается, *все* локальные переменные выходят из области видимости, и таким образом остаются установленными только те точки наблюдения, которые наблюдают за глобальными переменными. Если вы снова запустите программу, вы должны будете заново установить все

[1] Выражение не может быть реализовано с помощью точки наблюдения чтения/доступа. *(Прим. переводчика)*

[2] Аппаратная точка наблюдения *номер*: Не удается поместить точку наблюдения *(Прим. переводчика)*

такие точки наблюдения. Одним из способов сделать это будет установить точку останова на входе в функцию `main`, и, когда программа остановится, установить все точки наблюдения.

> *Предупреждение:* В многонитевых программах точки наблюдения являются лишь частично полезными. С текущей реализацией точек наблюдения, GDB может наблюдать только за величиной выражения *в одной нити*. Если вы уверены, что выражение может измениться только вследствие действий внутри текущей нити (и если вы также уверены, что никакая другая нить не может стать текущей), то вы можете использовать точки наблюдения как обычно. Однако, GDB может не заметить, когда действия в не текущей нити изменяют выражение.
>
> *Предупреждение для HP-UX:* В многонитевых программах, программные точки наблюдения являются лишь частично полезными. Если GDB создает программную точку наблюдения, она может наблюдать только за величиной выражения *в одной нити*. Если вы уверены, что выражение может измениться только вследствие действий внутри текущей нити (и если вы также уверены, что никакая другая нить не может стать текущей), то вы можете использовать программные точки наблюдения как обычно. Однако, GDB может не заметить, когда действия в не текущей нити изменяют выражение. (Аппаратные же точки наблюдения напротив, наблюдают за выражением во всех нитях.)

5.1.3 Установка точек перехвата

Вы можете использовать *точки перехвата*, чтобы вызвать остановку отладчика в ответ на определенные типы событий в программе, такие как исключения в Си++ или загрузка разделяемой библиотеки. Для установки точки перехвата используйте команду `catch`.

`catch` *событие*
: Остановиться, когда происходит *событие*. *Событие* может быть одним из:

  `throw` Выбрасывание исключения Си++.

  `catch` Перехват исключения Си++.

  `exec` Вызов `exec`. В настоящее время это доступно только на HP-UX.

  `fork` Вызов `fork`. В настоящее время это доступно только на HP-UX.

  `vfork` Вызов `vfork`. В настоящее время это доступно только на HP-UX.

  `load`
  `load` *имя-библ*
  : Динамическая загрузка любой разделяемой библиотеки, или загрузка библиотеки *имя-библ*. В настоящее время это доступно только на HP-UX.

  `unload`
  `unload` *имя-библ*
  : Выгрузка любой динамически загруженной разделяемой библиотеки, или выгрузка библиотеки *имя-библ*. В настоящее время это доступно только на HP-UX.

`tcatch` *событие*
: Установить точку перехвата, которая включена только до первой активации. Точка перехвата автоматически уничтожается после того, как событие перехвачено первый раз.

Используйте команду `info break` для получения списка текущих точек перехвата.

В настоящее время, в GDB существуют некоторые ограничения на обработку исключений Си++ (`catch throw` и `catch catch`):

- Если вы вызываете функцию интерактивно, GDB обычно возвращает вам управление после того, как функция закончила выполнение. Однако, если вызов возбуждает исключение, он может обойти механизм, возвращающий вам управление, и заставить вашу программу либо остановиться, либо просто продолжить выполнение до тех пор, пока она не активирует точку останова, получит сигнал, который ждет GDB, или выйдет. Это имеет место даже если вы установите точку перехвата для исключения; точки перехвата для исключений отключены при интерактивных вызовах.
- Вы не можете возбуждать исключения интерактивно.
- Вы не можете интерактивно установить обработчик исключения.

Иногда `catch` не является лучшим способом отладки обработки исключений: если вам необходимо точно знать, где исключение возбуждено, то лучше остановиться *до* того, как вызван обработчик исключения, так как в этом случае вы можете увидеть стек до того, как произойдет какое-либо развертывание. Если вместо этого вы установите точку останова в обработчике исключений, то может быть нелегко определить, где исключение было возбуждено.

Для остановки сразу перед вызовом обработчика исключений, вам необходимы некоторые знания о реализации. В случае GNU Си++, исключения возбуждаются путем вызова библиотечной функции `__raise_exception`, которая имеет следующий интерфейс ANSI Си:

```
/* addr - где хранится идентификатор исключения.
   id - идентификатор исключения.  */
void __raise_exception (void **addr, void *id);
```

Для того, чтобы отладчик перехватывал все исключения до того, как произойтет развертывание стека, установите точку останова на `__raise_exception` (см. Раздел 5.1 [Точки останова], с. 31).

С помощью условных точек останова (см. Раздел 5.1.6 [Условия останова], с. 40), зависящих от величины *id*, вы можете остановить вашу программу, когда возбуждается определенное исключение. Вы можете использовать несколько условных точек останова, чтобы остановить программу, когда возбуждается любое из нескольких исключений.

5.1.4 Удаление точек останова

Часто бывает необходимо уничтожить точку останова, наблюдения или перехвата, когда она сделала свое дело и вы больше не хотите останавливать там свою программу. Это называется *уничтожением* точки останова. Точка останова, которая была уничтожена, более не существует; она забыта.

С помощью команды `clear` вы можете удалять точки останова в соответствии с тем, где они находятся в вашей программе. С помощью команды `delete` вы можете удалять отдельные точки останова, наблюдения или перехвата, указывая их номера.

Не обязательно удалять точку останова, чтобы продолжить выполнение после нее. GDB автоматически игнорирует точки останова на первой инструкции, которая должна быть выполнена, когда вы продолжаете исполнение без изменения адреса выполнения.

`clear` Удаляет любые точки останова, установленные на следующей инструкции, которая должна быть выполнена в выбранном кадре стека (см. Раздел 6.3 [Выбор кадра], с. 52). Когда выбран самый внутренний кадр, это хороший способ удалить ту точку останова, на которой ваша программа только что остановилась.

`clear` *функция*
`clear` *имя-файла*:*функция*
Удалить любые точки останова, установленные на входе в *функцию*.

`clear` *номер-строки*
`clear` *имя-файла*:*номер-строки*
Удалить все точки останова, установленные на или внутри кода на указанной строке.

`delete` [`breakpoints`] [*диапазон*...]
Удалить точки останова, наблюдения или перехвата из диапазона, указанного в качестве аргумента. Если аргумент не задан, удалить все точки останова (GDB запрашивает подтверждение, если у вас не установлено `set confirm off`). Вы можете сократить это команду как `d`.

5.1.5 Отключение точек останова

Вместо того, чтобы удалять точку останова, наблюдения или перехвата, вам может быть удобнее *отключить* ее. Это делает точку останова бездействующей, как если бы она была удалена, но информация о ней запоминается, так что вы можете позже *включить* ее снова.

Вы отключаете и включаете точки останова, наблюдения и перехвата командами `enable` и `disable`, возможно указывая один или более номеров точек останова в качестве аргументов. Используйте `info break` или `info watch` для распечатки списка точек останова, наблюдения и перехвата, если вы не знаете какие номера использовать.

Точка останова, наблюдения или перехвата может находиться в одном из четырех состояний:

- Включена. Точка останова останавливает вашу программу. Точка останова, установленная командой `break`, изначально находится в таком состоянии.
- Отключена. Точка останова не оказывает воздействия на вашу программу.
- Включена до первого срабатывания. Точка останова останавливает вашу программу, но потом становится отключенной.
- Включена для удаления. Точка останова останавливает вашу программу, но сразу после этого она удаляется навсегда. Точка останова, установленная командой `tbreak`, изначально находится в этом состоянии.

Вы можете использовать следующие команды для включения или отключения точек останова, наблюдения и перехвата:

`disable` [`breakpoints`] [*диапазон*...]
Отключить указанные точки останова, или все точки останова, если ни одна не перечислена. Отключенная точка останова не оказывает никакого действия, но она не забывается. Все параметры, такие как счетчик игнорирований, условия и команды запоминаются, на случай, если точка останова позже будет снова включена. Вы можете сокращать `disable` как `dis`.

`enable` [`breakpoints`] [*диапазон*...]
Включает указанные (или все определенные) точки останова. Они снова становятся значимыми для остановки вашей программы.

`enable` [`breakpoints`] `once` *диапазон*...
Временно включить указанные точки останова. GDB отключает любую из этих точек останова немедленно после срабатывания.

`enable` [`breakpoints`] `delete` *диапазон*...
: Включить указанные точки останова до первого срабатывания, затем уничтожить. GDB удаляет любую из этих точек останова, как только ваша программа останавливается на ней.

Кроме точек останова, установленных командой `tbreak` (см. Раздел 5.1.1 [Установка точек останова], с. 32), установленные вами точки останова изначально включены; следовательно, они становятся отключенными или включенными только когда вы используете одну из вышеперечисленных команд. (Команда `until` может устанавливать и удалять свою собственную точку останова, но она не изменяет состояние ваших других точек останова; см. Раздел 5.2 [Продолжение и выполнение по шагам], с. 44.)

5.1.6 Условия останова

Простейшая точка останова останавливает вашу программу каждый раз, когда управление достигает заданного места. Вы можете также указать *условие* для точки останова. Условие является просто булевым выражением в вашем языке программирования (см. Раздел 8.1 [Выражения], с. 61). Точка останова с условием вычисляет выражение каждый раз, когда ваша программа достигает ее, и ваша программа остановится только в том случае, если условие *истинно*.

Это противоположно использованию утверждений для проверки правильности программы; в этом случае, вы хотите остановиться, когда утверждение нарушается—то есть, когда условие ложно. В Си, если вы хотите проверить утверждение, выраженное условием *assert*, вы должны установить условие '`!` *assert*' на соответствующей точке останова.

Условия также допускаются для точек наблюдения; вам они могут не понадобиться, так как точка наблюдения так или иначе контролирует значение выражения—но может оказаться проще, скажем, просто установить точку наблюдения на имя переменной и указать условие, проверяющее, является ли новое значение тем, которое нас интересует.

Условия останова могут иметь побочные эффекты, и даже могут вызывать функции в вашей программе. Это может быть полезным, например, для активации функций, которые запоминают продвижение выполнения вашей программы, или для использования ваших собственных функций печати для форматирования специальных структур данных. Результаты полностью предсказуемы, если нет другой включенной точки останова по тому же адресу. (В этом случае, GDB может сначала увидеть другую точку останова и остановить вашу программу программу без проверки условия первой точки останова.) Заметьте, что команды точек останова обычно более удобны и гибки, чем условия останова, для выполнения побочных эффектов, когда достигается точка останова (см. Раздел 5.1.7 [Команды точки останова], с. 41).

Условия останова могут быть заданы в момент установки точки останова, используя '`if`' в аргументах команды `break`. См. Раздел 5.1.1 [Установка точек останова], с. 32. Они могут быть также изменены в любой момент с помощью команды `condition`.

Вы также можете использовать ключевое слово `if` с командой `watch`. Команда `catch` не распознает ключевое слово `if`; `condition` является единственным способом наложить дальнейшие условия на точку перехвата.

`condition` *номер выражение*
: Задайте *выражение* как условие остановки для точки останова, наблюдения или перехвата с номером *номер*. После того, как вы установили условие, данная точка останова остановит вашу программу только если значение *выражения* будет истинным (ненулевым, в Си). Когда вы используете `condition`, GDB немедленно проверяет *выражение* на синтаксическую корректность и для определения, что символы в нем имеют объекты ссылки в контексте вашей точки останова. Если *выражение* использует символы, не существующие в контексте точки останова, GDB выведет сообщение об ошибке:

```
No symbol "foo" in current context.
```
[3]

Однако, GDB в действительности не вычисляет *выражение* в момент подачи команды `condition` (или команды, устанавливающей точку останова с условием, такой как `break if ...`). См. Раздел 8.1 [Выражения], с. 61.

`condition` *номер*
: Снимает условие с точки останова с номером *номер*. Она становится обычной безусловной точкой останова.

Специальным случаем условия для точки останова является остановка только когда точка останова была достигнута определенное число раз. Это настолько полезно, что существует специальный способ сделать это, используя *счетчик игнорирования* точки останова. Каждая точка останова имеет счетчик игнорирования, являющийся целым числом. Как правило, счетчик игнорирования равен нулю, и, следовательно, не производит никакого действия. Но если ваша программа достигает точки останова, чей счетчик игнорирования положителен, тогда вместо того чтобы остановиться, она лишь уменьшит его на единицу и продолжит выполнение. В результате, если величина счетчика игнорирования равна *n*, точка останова не остановит программу следующие *n* раз, когда программа его достигнет.

`ignore` *номер значение*
: Устанавливает счетчик игнорирований точки останова с номером *номер* в *значение*. Следующие *значение* раз, когда точка останова будет достигнута, выполнение вашей программы не будет остановлено; кроме как уменьшить счетчик игнорирований, GDB не производит никаких действий.

  Чтобы точка останова сработала при следующем достижении, установите счетчик в ноль.

  Когда вы используете `continue` для возобновления выполнения вашей программы от точки останова, вы можете установить счетчик игнорирований непосредственно как аргумент к `continue`, а не использовать `ignore`. См. Раздел 5.2 [Продолжение и выполнение по шагам], с. 44.

  Если точка останова имеет положительный счетчик игнорирований и условие, то условие не проверяется. Как только счетчик игнорирований достигнет нуля, GDB возобновит проверку условия.

  Вы можете достигнуть эффекта счетчика игнорирований с помощью такого условия, как '`$foo- <= 0`', используя вспомогательную переменную отладчика, которая уменьшается каждый раз. См. Раздел 8.9 [Вспомогательные переменные], с. 73.

Счетчики игнорирований можно использовать с точками останова, точками наблюдения и точками перехвата.

5.1.7 Команды точки останова

Вы можете подать любой точке останова (наблюдения или перехвата) ряд команд, которые будут выполняться при остановке вашей программы на этой точке останова. Например, вы можете захотеть вывести значения определенных выражений, или включить другие точки останова.

`commands` [*номер*]
... список-команд ...
`end`
: Определяет список команд для точки останова с номером *номер*. Сами команды указываются в следующих строках. Для завершения списка команд, введите строку, содержащую только `end`.

[3] В текущем контексте нет символа "foo". *(Прим. переводчика)*

Чтобы удалить все команды от точки останова, введите `commands` и немедленно за этим `end`, то есть задайте пустой список команд.

Без аргумента *номер*, `commands` относится к последней установленной точке останова, наблюдения или перехвата (но не к последней встреченной).

Нажатие ⟨RET⟩, как средство повторения последней команды GDB, отключено внутри *списка-команд*.

Вы можете использовать команды для точки останова, чтобы снова запустить вашу программу на выполнение. Просто используйте команду `continue`, или `step`, или любую другую команду, возобновляющую выполнение.

После команды, возобновляющей выполнение, любые другие команды в списке игнорируются. Так сделано потому, что каждый раз, когда вы возобновляете выполнение (даже просто с помощью `next` или `step`), вы можете встретить другую точку останова—которая может иметь свой собственный список команд, что приведет к неоднозначности, какой из списков выполнять.

Если в качестве первой команды в списке команд вы укажете `silent`, обычное сообщение об остановке на точке останова не будет выводиться. Это может быть желательно для точек останова, которые должны вывести определенное сообщение, и затем продолжить выполнение. Если никакая из оставшихся команд ничего не выводит, вы не увидите никакого знака о том, что точка останова была достигнута. `silent` имеет смысл только в начале списка команд точки останова.

Команды `echo`, `output` и `printf` позволяют вам более точно контролировать выводимый текст, и часто полезны в “тихих” точках останова. См. Раздел 16.4 [Команды управления выводом], с. 159.

Например, вот как вы можете использовать команды точки останова для вывода величины `x` на входе в `foo`, когда `x` положительна.

```
break foo if x>0
commands
silent
printf "x is %d\n",x
cont
end
```

Одним из применений команд точки останова является компенсация одной ошибки, так, чтобы вы могли искать другую. Поместите точку останова сразу после строки кода, содержащей ошибку, задайте ей условие для определения случая, в котором было сделано что-то ошибочное, и определите команды для присвоения правильных значений тем переменным, для которых это требуется. Закончите командой `continue`, чтобы ваше программа не останавливалась, а начните с команды `silent`, чтобы не было никакого вывода. Вот пример:

```
break 403
commands
silent
set x = y + 4
cont
end
```

5.1.8 Меню точки останова

Некоторые языки программирования (особенно Си++) допускают, чтобы одно и то же имя функции было определено несколько раз, для применения в различных контекстах. Это называется *перегрузкой*. Когда имя функции перегружается, ‘`break` *функция*’ не достаточно, чтобы указать GDB, где вы хотите установить точку останова. Если вы столкнулись

с этой проблемой, вы можете использовать что-то типа ‘`break` *функция*`(`*типы*`)`’ для указания, какую конкретную версию функции вы имеете в виду. В противном случае, GDB предлагает вам выбор из пронумерованных вариантов для различных возможных точек останова, и ждет вашего выбора с приглашением ‘`>`’. Первыми двумя вариантами всегда являются ‘`[0] cancel`’ и ‘`[1] all`’. Ввод *1* устанавливает точку останова на каждом определении *функции*, и ввод *0* прерывает команду `break` без установки новых точек останова.

Например, следующая выдержка из сеанса иллюстрирует попытку установить точку останова на перегруженном символе `String::after`. Мы выбрали три конкретных определения имени функции:

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
 breakpoints.
(gdb)
```

5.1.9 “Не удается поместить точки останова”

В некоторых операционных системах точки останова не могут быть использованы в программе, если какой-либо другой процесс выполняет эту программу. В этом случае, попытка выполнить или продолжить выполнение программы с точкой останова приводит тому, что GDB печатает сообщение об ошибке:

```
Cannot insert breakpoints.
The same program may be running in another process.
```
[4]

Когда это происходит, у вас есть три варианта дальнейших действий:

1. Удалить или отключить точки останова, и затем продолжить.
2. Приостановить GDB и скопировать файл, содержащий вашу программу, под другим иненем. Возобновить работу GDB и использовать команду `exec-file` для указания, что GDB должен выполнять вашу программу под этим именем. Затем запустите вашу программу снова.
3. Скомпоновать заново вашу программу так, чтобы сегмент текста был неразделяемым, используя ключ компоновщика ‘`-N`’. Ограничения операционной системы могут не распространяться на неразделяемые выполняемые файлы.

Аналогичное сообщение может выводиться, если вы запрашиваете слишком много активных аппаратно-поддерживаемых точек останова и наблюдения:

```
Stopped; cannot insert breakpoints.
```

[4] Не удается поместить точки останова. Эта программа может выполняться в другом процессе. *(Прим. переводчика)*

`You may have requested too many hardware breakpoints and watchpoints.`[5]

Это сообщение выводится, когда вы пытаетесь возобновить выполнение программы, так как только тогда GDB знает точно, сколько аппаратных точек останова и наблюдения ему нужно установить.

Когда это сообщение выводится, вам необходимо отключить или удалить некоторые аппаратно-поддерживаемые точки останова и наблюдения, и затем продолжить.

5.2 Продолжение и выполнение по шагам

Продолжение означает возобновление выполнения программы до ее нормального завершения. Напротив, *пошаговое выполнение* означает выполнение еще одного "шага" вашей программы, где "шаг" быть либо одной строкой исходного кода, либо одной машинной инструкцией (в зависимости от того, какую именно команду вы используете). И в случае продолжения, и в случае выполнения по шагам, ваша программа может остановиться и раньше, вследствие точки останова или сигнала. (Если она останавливается по сигналу, вы можете использовать `handle`, или '`signal 0`' для возобновления выполнения. См. Раздел 5.3 [Сигналы], с. 46.)

`continue` [*счетчик-игнор*]
`c` [*счетчик-игнор*]
`fg` [*счетчик-игнор*]

Возобновить выполнение программы, с того адреса, где ваша программа остановилась последний раз; все точки останова, установленные по этому адресу, пропускаются. Необязательный аргумент *счетчик-игнор* позволяет вам задать количество последующих игнорирований точки останова в этом месте; его действие совпадает с действием `ignore` (см. Раздел 5.1.6 [Условия останова], с. 40).

Аргумент *счетчик-игнор* имеет смысл только если ваша программа остановилась в точке останова. В остальных случаях, аргумент к `continue` игнорируется.

Синонимы `c` и `fg` (от *foregroung*, так как отлаживаемая программа считается фоновой), предоставляются исключительно для удобства, и имеют в точности тот же смысл, что и `continue`.

Чтобы возобновить выполнение с другого места, вы можете использовать `return` (см. Раздел 11.4 [Возврат из функции], с. 103) чтобы вернуться назад к вызывающей функции; или `jump` (см. Раздел 11.2 [Продолжение с другого адреса], с. 102) для перехода к произвольному месту в вашей программе.

Типичная техника для использования пошагового выполнения заключается в установке точки останова (см. Раздел 5.1 [Точки останова], с. 31) на начале функции или раздела вашей программы, где предположительно находится ошибка, выполнении вашей программы до остановки на этой точке останова, и затем пошаговом выполнении подозреваемого участка, с исследованием интересуемых переменных, пока вы не увидите, что случилась ошибка.

`step` Продолжить выполнение вашей программы, пока управление не достигнет другой строки исходного текста, затем остановить ее и возвратить управление GDB. Эту команду можно сокращать до `s`.

Предупреждение: Если вы используете команду `step`, когда управление находится внутри функции, которая была скомпилирована без

[5] Остановлено; не удается поместить точки останова. Вы могли запросить слишком много аппаратно-поддерживаемых точек останова и наблюдения. *(Прим. переводчика)*

отладочной информации, выполнение продолжается, пока управление не достигнет функции, которая *имеет* ее. Аналогично, пошаговое выполнение не будет заходить в функцию, скомпилированную без отладочной информации. Для пошагового выполнения таких функций используйте команду `stepi`, описанную ниже.

Команда `step` останавливается только на первой инструкции строки исходного текста. Это предотвращает множественные остановки, которые в противном случае могут возникнуть в операторе `switch`, цикле `for`, и так далее. `step` продолжает останавливаться, если функция, имеющая отладочную информацию, вызывается внутри строки. Другими словами, `step` *заходит внутрь* функций, вызываемых в данной строке.

Также, команда `step` входит в функцию только если для нее существует информация о номерах строк. Иначе она действует как команда `next`. Это позволяет избежать проблем, появляющихся при использовании `cc -gl` на машинах MIPS. Раньше `step` заходила в подпрограмму, если существовала хоть какая-нибудь отладочная информация о подпрограмме.

`step` *число*

Продолжает выполнение как по команде `step`, но делает это *число* раз. Если достигается точка останова, или приходит сигнал, не связанный с пошаговым выполнением, до выполнения *числа* шагов, пошаговое выполнение сразу останавливается.

`next` [*число*]

Продолжает выполнение до следующей строки исходного текста в текущем (внутреннем) кадре стека. Это аналогично `step`, но вызовы функций, которые появляются внутри строки кода, выполняются без остановки. Выполнение останавливается, когда управление достигает другой строки кода в исходном уровне стека, который выполнялся, когда вы дали команду `next`. Эта команда сокращается как `n`.

Аргумент *число* является счетчиком повторений, как для `step`.

Команда `next` останавливается только на первой инструкции исходной строки. Это предотвращает множественные остановки, которые иначе могут возникнуть в операторах `switch`, циклах `for`, и так далее.

`finish` Продолжить выполнение до возврата из функции в выбранном кадре стека. Напечатать возвращенное значение (если таковое существует).

Сравните это с командой `return` (см. Раздел 11.4 [Возврат из функции], с. 103).

`until`
`u` Продолжить выполнение до достижения строки исходного текста, следующей за текущей, в текущем кадре стека. Эта команда используется для избежания выполнения цикла по шагам больше одного раза. Она похожа на команду `next`, за исключением того, что когда `until` встречает переход, она автоматически продолжает выполнение, пока счетчик выполнения программы не станет больше, чем адрес перехода.

Это означает, что когда вы достигаете конца цикла после его выполнения по шагам, `until` продолжает выполнение вашей программы, пока она не выйдет из цикла. Напротив, команда `next` в конце цикла просто переходит назад в начало цикла, что заставляет вас выполнять по шагам следующую итерацию.

`until` всегда останавливает вашу программу, если она пытается выйти из текущего кадра стека.

`until` может привести к несколько неожиданным результатам, если порядок машинных кодов не совпадает с порядком строк исходного текста. Например,

в следующем отрывке сеанса отладки, команда `f` (`frame`) показывает, что выполнение остановилось на строке `206`; хотя, когда мы используем `until`, мы переходим к строке `195`:

```
(gdb) f
#0  main (argc=4, argv=0xf7fffae8) at m4.c:206
206                 expand_input();
(gdb) until
195             for ( ; argc > 0; NEXTARG) {
```

Это произошло потому, что для эффективности выполнения компилятор сгенерировал код для проверки окончания цикла в конце, а не в начале цикла—даже если проверка в цикле `for` Си написана до тела цикла. Кажется, что команда `until` переместилась назад к началу цикла, когда двигалась к этому выражению; однако, в действительности она не переходила к более раннему оператору—в терминах фактического машинного кода.

`until` без аргументов работает посредством пошагового выполнения отдельных инструкций, и, следовательно, является более медленной, чем `until` с аргументом.

`until` *положение*
`u` *положение*

Продолжить выполнение вашей программы, пока либо указанное место не будет достигнуто, либо не произойдет возврат из текущего кадра стека. *положение* может быть любой из доступных форм аргумента для `break` (см. Раздел 5.1.1 [Установка точек останова], с. 32). Эта форма команды использует точки останова, и, следовательно, является более быстрой, чем `until` без аргумента.

`stepi`
`stepi` *арг*
`si`

Выполнить одну машинную инструкцию, затем остановиться и вернуться в отладчик.

При пошаговом выполнении машинных инструкций, часто бывает полезным сделать ‘`display/i $pc`’. Это велит GDB автоматически отображать инструкцию, которая будет выполняться следующей, каждый раз, когда ваша программа останавливается. См. Раздел 8.6 [Автоматическое отображение], с. 66.

Аргумент является счетчиком повторений, как для `step`.

`nexti`
`nexti` *арг*
`ni`

Выполнить одну машинную инструкцию, но если это вызов функции, продолжать до возврата из нее.

Аргумент является счетчиком повторений, как для `next`.

5.3 Сигналы

Сигнал—это асинхронное событие, которое может произойти в программе. Операционная система определяет возможные типы сигналов и дает каждому типу имя и номер. В Unix, например, `SIGINT`—это сигнал, получаемый программой, когда вы вводите знак прерывания (часто *C-c*); `SIGSEGV`—сигнал, получаемый программой при ссылке на область памяти, отличную от всех используемых областей; `SIGALRM` появляется при срабатывании интервального таймера (возникает только, если ваша программа запросила временной сигнал).

Некоторые сигналы, такие как `SIGALRM`, являются обычной частью функционирования вашей программы. Другие, такие как `SIGSEGV`, обозначают ошибки; эти сигналы являются

фатальными (они немедленно убивают вашу программу), если программа не определила заранее другой способ их обработки. `SIGINT` не указывает на ошибку в вашей программе, но обычно является фатальным, так что он может выполнять функцию прерывания: убить программу.

GDB способен обнаружить любое появление сигнала в вашей программе. Вы можете заранее сообщить GDB, что делать для каждого типа сигнала.

Обычно, GDB установлен так, чтобы игнорировать неошибочные сигналы, такие как `SIGALRM` (чтобы не мешать их действию при исполнении вашей программы), но немедленно останавливать вашу программу всякий раз, когда возникает сигнал об ошибке. Вы можете изменить эти установки командой `handle`.

`info signals`
`info handle`
: Напечатать таблицу всех типов сигналов и описания, как GDB будет обрабатывать каждый из них. Вы можете использовать эту команду, чтобы посмотреть номера всех определенных типов сигналов.

  `info handle` является синонимом для `info signals`.

`handle` *сигнал ключевые-слова...*
: Изменить способ, которым GDB обрабатывает *сигнал*. *сигнал* может быть номером сигнала или его именем (с '`SIG`' или без него в начале). *Ключевые-слова* определяют, какие сделать изменения.

Ключевые слова, допускаемые командой `handle`, могут быть сокращены. Вот их полные имена:

`nostop`
: GDB не должен останавливать вашу программу при получении этого сигнала. Все же он может вывести сообщение, уведомляющее о получении сигнала.

`stop`
: GDB должен остановить вашу программу при получении этого сигнала. Это также подразумевает ключевое слово `print`.

`print`
: GDB должен вывести сообщение при возникновении данного сигнала.

`noprint`
: GDB вообще не должен замечать возникновение сигнала. Это также подразумевает ключевое слово `nostop`.

`pass`
: GDB должен позволить вашей программе увидеть этот сигнал; ваша программа может обработать сигнал, или же она может завершиться, если сигнал фатальный и не обработан.

`nopass`
: GDB не должен позволять вашей программе видеть этот сигнал.

Когда сигнал останавливает вашу программу, он невидим для нее, пока вы не продолжите выполнение. Затем ваша программа видит сигнал, если *в данный момент* на рассматриваемый сигнал распространяется действие команды `pass`. Другими словами, после того, как GDB сообщит о сигнале, вы можете использовать команду `handle` с `pass` или `nopass`, чтобы указать, должна ли ваша программа увидеть этот сигнал при продолжении.

Вы также можете использовать команду `signal` для того, чтобы помешать вашей программе увидеть сигнал или, наоборот, заставить ее заметить обычно игнорируемый сигнал, или чтобы подать ей произвольный сигнал в любое время. Например, если ваша программа остановилась вследствие какой-либо ошибки обращения к памяти, вы можете сохранить правильные значения в ошибочные переменные и продолжить выполнение, в надежде посмотреть на дальнейшее выполнение, но ваша программа вероятно немедленно остановилась бы из-за фатального сигнала, как только она бы его заметила. Чтобы помешать этому, вы можете продолжить выполнение с '`signal 0`'. См. Раздел 11.3 [Подача сигнала вашей программе], с. 103.

5.4 Остановка и запуск многонитевых программ

Когда ваша программа имеет несколько нитей выполнения (см. Раздел 4.9 [Отладка многонитевых программ], с. 26), вы можете выбрать, установить точки останова либо во всех, либо в каких-то отдельных нитях.

`break` *ном-строки* `thread` *номер-нити*
`break` *ном-строки* `thread` *номер-нити* `if ...`

> *ном-строки* определяет строки исходного текста; существует несколько способов их задания, но результат всегда один и тот же—указать строку исходного текста.
>
> Используйте классификатор '`thread` *номер-нити*' с командой точки останова, чтобы указать GDB, что вы хотите остановить программу, только когда определенная нить достигнет этой точки. *номер-нити*—это один из числовых идентификаторов нити, присвоенный GDB, показываемый в первой колонке при выводе '`info threads`'.
>
> Если при установке точки останова вы не укажете '`thread` *номер-нити*', точка останова будет действовать для *всех* нитей вашей программы.
>
> Вы также можете использовать классификатор `thread` для условных точек останова; в этом случае, поместите '`thread` *номер-нити*' перед условием точки останова, вот так:
>
> ```
> (gdb) break frik.c:13 thread 28 if bartab > lim
> ```

При любой остановке вашей программы под управлением GDB, прекращается выполнение *всех* нитей, а не только текущей. Это позволяет вам исследовать полное состояние программы, включая переключение между нитями, не опасаясь, что это может изменить что-либо в дальнейшем.

Наоборот, когда вы снова запускаете программу, *все* нити начинают выполняться. *Это верно даже при пошаговом выполнении* такими командами, как `step` или `next`.

В частности, GDB не может пошагово выполнять все нити параллельно. Так как планированием выполнения нити занимается операционная система отлаживаемой цели (не контролируемая GDB), то пока в текущей нити выполняется один шаг, в других может выполниться несколько. Более того, когда выполнение программы останавливается, другие потоки вообще могут остановиться в середине операторов, а не на границе между ними.

Вы даже можете обнаружить, что после продолжения исполнения или после пошагового выполнения ваша программа остановилась в другой нити. Это случается всякий раз, когда другая нить достигает точки останова, получает сигнал или в ней возникает исключительная ситуация, прежде чем первая нить завершает выполнение того, что вы запросили.

В некоторых операционных системах вы можете заблокировать планировщик заданий и тем самым позволить выполняться только одной нити.

`set scheduler-locking` *режим*

> Устанавливает режим блокировки планировщика заданий. Если он установлен в `off`, то блокировки нет и любая нить может выполняться в любое время. Если этот режим установлен в `on`, то только текущая нить может выполняться, когда выполнение продолжается. Режим `step` производит оптимизацию для пошагового выполнения. Он не дает другим нитям "захватывать приглашение" путем приоритетного прерывания обслуживания текущей нити во время пошагового выполнения. Другие нити едва ли получат возможность начать выполнение, когда вы выполняете очередной шаг. С большей вероятностью они начнут выполняться, когда вы выполняете команду `next` на вызове функции, и

им не что не помешает выполняться, когда вы используете такие команды, как ‘`continue`’, ‘`until`’ или ‘`finish`’. Однако, если другие нити не достигнут точки останова в течение отведенного ему для выполнения времени, они никогда не перехватят приглашение GDB у отлаживаемой вами нити.

`show scheduler-locking`
: Отобразить текущий режим блокировки.

6 Исследование стека

Когда ваша программа остановилась, первое, что вам нужно знать—где она остановилась и как она туда попала.

Каждый раз, когда ваша программа производит вызов функции, о нем создается определенная информация. Она включает положение вызова в вашей программе, параметры вызова и локальные переменные вызываемой функции. Информация сохраняется в блоке данных, называемом *кадром стека*. Кадры стека размещаются в области памяти, называемой *стеком вызовов*.

Команды GDB для исследования стека позволяют вам увидеть всю эту информацию при остановке вашей программы.

Один из кадров стека является *выбранным* GDB, и многие команды GDB неявно относятся к нему. В частности, когда вы запрашиваете у GDB значение переменной вашей программы, это значение находится в выбранном кадре. Для выбора интересующего вас кадра существуют специальные команды GDB. См. Раздел 6.3 [Выбор кадра], с. 52.

Когда ваша программа останавливается, GDB автоматически выбирает текущий выполняющийся кадр и выдает его краткое описание, аналогично команде `frame` (см. Раздел 6.4 [Информация о кадре стека], с. 53).

6.1 Кадры стека

Стек вызовов разделен на непрерывные участки, называемые *кадрами стека*, или *кадрами* для краткости; каждый кадр является данными, связанными с одним вызовом одной функции. Кадр содержит аргументы, переданные функции, ее локальные переменные и адрес, с которого она выполняется.

Когда ваша программа стартует, стек содержит только один кадр—для функции `main`. Он называется *начальным* или *внешним* кадром. Каждый раз при вызове функции создается новый кадр. При каждом выходе из функции, кадр этого вызова функции уничтожается. Если функция является рекурсивной, для нее может существовать множество кадров. Кадр для функции, исполняемой в данный момент, называется *внутренним* кадром. Это кадр, созданный самым последним из всех существующих кадров стека.

Внутри вашей программы кадры стека идентифицируются своим адресом. Кадр стека состоит из множества байт, каждый из которых имеет свой собственный адрес; каждый тип компьютеров имеет свой способ для выбора одного байта, чей адрес служит адресом кадра. Обычно, пока выполнение происходит в данном кадре, этот адрес содержится в регистре, называемом *регистром указателя кадра*.

GDB присваивает номера всем существующим кадрам стека, начиная с нуля для внутреннего кадра, единицу—вызвавшему его кадру, и так далее. В действительности, эти номера не существуют в вашей программе; они назначаются GDB, чтобы предоставить вам способ различать кадры стека в командах GDB.

Некоторые компиляторы позволяют компилировать функции так, чтобы они выполнялись без создания кадров стека. (Например, ключ gcc

```
'-fomit-frame-pointer'
```

создает функции без кадра.) Это иногда делается с часто используемыми библиотечными функциями, чтобы сохранить время, требуемое для установки кадра. GDB имеет ограниченные возможности для обработки таких функциональных вызовов. Если вызов внутренней функции происходит без создания кадра стека, GDB, тем не менее, описывает его так, как если бы он имел отдельный кадр, который имеет, как обычно, номер 0, позволяя корректно трассировать цепочку функциональных вызовов. Однако, GDB не имеет средств для работы с функциями без кадра в другом месте стека.

`frame` *apr* Команда `frame` позволяет вам перемещаться от одного кадра стека к другому, и распечатывать выбранный вами кадр. *Apr* может быть либо адресом кадра, либо его номером. Без аргумента, `frame` выводит текущий кадр стека.

`select-frame`
Команда `select-frame` позволяет вам перемещаться от одного кадра стека к другому без его распечатки. Это "тихая" версия `frame`.

6.2 Цепочки вызовов

Цепочка вызовов предоставляет собой информацию о том, как ваша программа оказалась там, где она есть. Она отображает по одной строке для каждого кадра, начиная с текущего выполняющегося кадра (кадра 0), за которым следует кадр, из которого он был вызван (кадр 1), и далее вверх по стеку.

`backtrace`

`bt` Вывести цепочку вызовов всего стека: по одной строке на кадр, для всех кадров в стеке.

Вы можете прервать цепочку вызовов в любое время, введя знак системного прерывания, обычно `C-c`.

`backtrace` *n*

`bt` *n* То же самое, но выводятся только *n* внутренних кадров.

`backtrace -`*n*

`bt -`*n* То же самое, но выводятся только *n* внешних кадров.

`where` и `info stack` (сокращенно `info s`)—дополнительные синонимы для `backtrace`.

Каждая строка в цепочке вызовов показывает номер кадра и имя функции. Счетчик команд также показывается, если только вы не используете `set print address off`. Цепочка вызовов также показывает имя исходного файла, номер строки и аргументы функции. Значение счетчика команд опускается, если он указывает на начало кода для данной строки.

Ниже приведен пример цепочки вызовов. Она была получена командой '`bt 3`', так что она показывает три внутренних кадра.

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
(More stack frames follow...)
```

Информация о нулевом кадре не начинается со значения счетчика команд, что указывает на то, что ваша программа остановилась в начале кода для строки `993` файла `builtin.c`.

6.3 Выбор кадра

Большинство команд для исследования стека и других данных в вашей программе применяются выбранному в данный момент кадру. Здесь приведены команды для выбора кадра стека; все они завершаются выводом краткого описания выбранного кадра стека.

`frame` *n*

`f` *n* Выбрать кадр номер *n*. Напоминаем, что нулевой кадр—это внутренний (исполняемый в данный момент) кадр, первый кадр—тот, из которого вызван нулевой, и так далее. Кадр с наибольшим номером—это кадр для функции `main`.

`frame` *адрес*
`f` *адрес* Выбрать кадр, расположенный по адресу *адрес*. В основном это полезно, если формирование цепочки кадров стека было нарушено из-за ошибки, сделавшей невозможным для GDB правильное присвоение номеров всем кадрам. Кроме того, это может быть полезным, когда у вашей программы есть несколько стеков и происходит переключение от одного к другому.

В архитектуре SPARC, команде `frame` для выбора произвольного кадра необходимо указать два адреса: указатель кадра и указатель вершины стека.

В архитектурах MIPS и Alpha требуется два адреса: указатель вершины стека и указатель команд.

В архитектуре 29k требуется три адреса: указатель вершины стека регистров, указатель команд и указатель вершины стека памяти.

`up` *n* Переместиться вверх по стеку на *n* кадров. Для положительных значений *n*, это перемещение происходит по направлению к внешнему кадру, к кадрам с большими номерами, к кадрам, которые существуют дольше. По умолчанию *n* принимается равным единице.

`down` *n* Передвинуться вниз по стеку на *n* кадров. Для положительных значений *n*, это продвижение происходит по направлению к внутреннему кадру, к кадру с меньшим номером, к кадрам, которые были созданы позже. По умолчанию, значение *n* принимается равным единице. Вы можете сокращать `down` как `do`.

Все эти команды заканчиваются выводом двух строк, описывающих кадр. Первая строка показывает номер кадра, имя функции, аргументы, имя исходного файла и номер выполняемой строки в этом кадре. Вторая строка показывает содержимое этой строки исходного текста.

Например:

```
(gdb) up
#1  0x22f0 in main (argc=1, argv=0xf7fffbf4, env=0xf7fffbfc)
    at env.c:10
10              read_input_file (argv[i]);
```

После такого вывода, команда `list` без аргументов выводит десять строк, расположенных вокруг точки выполнения в кадре. См. Раздел 7.1 [Вывод строк исходного текста], с. 55.

`up-silently` *n*
`down-silently` *n*
Эти две команды являются вариантами `up` и `down` соответственно, отличаясь от них тем, что делают свою работу “тихо”, не отображая новый кадр. Они предназначены для использования в основном в командных сценариях GDB, где вывод может быть ненужным и отвлекающим.

6.4 Информация о кадре стека

Существуют несколько других команд для вывода информации о выбранном кадре стека.

`frame`
`f` При использовании без аргументов, эта команда не выбирает новый кадр, а выводит краткое описание текущего выбранного кадра стека. Эту команду можно сокращать как `f`. С аргументом, эта команда используется для выбора кадра стека. См. Раздел 6.3 [Выбор кадра стека], с. 52.

`info frame`
`info f` Эта команда выводит подробное описание выбранного кадра стека, включающее:

- адрес кадра
- адрес следующего вниз по стеку кадра (вызываемого из данного)
- адрес следующего вверх по стеку кадра (того, из которого был вызван данный)
- язык, на котором написан исходный код, соответствующий этому кадру
- адрес аргументов кадра
- адрес локальных переменных кадра
- сохраненный в кадре счетчик команд (адрес выполнения в кадре, вызвавшем данный)
- регистры, которые были сохранены в кадре

Подробное описание полезно, если из-за какой-либо ошибки формат стека не соответствует обычным соглашениям.

`info frame` *адрес*
`info f` *адрес*
Вывести подробное описание кадра стека, расположенного по адресу *адрес*, не выбирая этот кадр. Выбранный кадр этой командой не изменяется. Она требует параметр *адрес* того же типа, что и команда `frame` (для некоторых архитектур не один, а несколько). См. Раздел 6.3 [Выбор кадра стека], с. 52.

`info args` Вывести аргументы выбранного кадра, каждый на отдельной строке.

`info locals`
Вывести локальные переменные выбранного кадра, каждую на отдельной строке. Выводятся все переменные (объявленные как статические или как автоматические), доступные в точке выполнения выбранного кадра стека.

`info catch`
Выводит список всех обработчиков исключительных ситуаций, являющихся активными в текущей точке выполнения текущего кадра стека. Чтобы увидеть другие обработчики исключительных ситуаций, перейдите в соответствующую секцию (используя команды `up`, `down` или `frame`); затем наберите `info catch`. См. Раздел 5.1.3 [Установка точек перехвата], с. 37.

7 Исследование исходных файлов

GDB может выводить части исходных текстов вашей программы, так как отладочная информация, записанная в ней, сообщает GDB, какие исходные файлы использовались при создании программы. Когда ваша программа останавливается, GDB сам выводит строку, на которой она остановилась. Аналогично, когда вы выбираете кадр стека (см. Раздел 6.3 [Выбор кадра стека], с. 52), GDB выводит строку, на которой остановилось выполнение в этом кадре. Вы можете выводить другие части исходных файлов с помощью явных команд.

Если вы используете GDB через интерфейс к GNU Emacs, вы можете предпочесть воспользоваться средствами Emacs для просмотра исходных текстов; смотрите Глава 17 [Использование GDB под управлением GNU Emacs], с. 161.

7.1 Вывод строк исходного текста

Чтобы вывести строки файла с исходным текстом, используйте команду `list` (сокращенно `l`). По умолчанию выводятся десять строк. Существует несколько способов определения того, какую часть файла вы хотите вывести.

Здесь представлены наиболее употребительные формы команды `list`:

`list` *номер-строки*
: Вывести строки, расположенные вокруг строки с номером *номер-строки* в текущем исходном файле.

`list` *функция*
: Вывести строки, расположенные вокруг начала функции *функция*.

`list`
: Вывести еще определенное количество строк. Если последние выведенные строки выводились с помощью команды `list`, то выводятся строки, следующие за последними выведенными; если, однако, последней выведенной строкой была одиночная строка, выведенная как часть отображения кадра стека (см. Глава 6 [Исследование стека], с. 51), то выводятся строки, расположенные вокруг нее.

`list -`
: Вывести строки, расположенные непосредственно перед последними выведенными.

По умолчанию, для любой из этих форм команды `list` GDB выводит десять строк исходного текста. Вы можете изменить это командой `set listsize`:

`set listsize` *число*
: Установить количество выводимых командой `list` строк в *число* (если аргумент команды `list` не задает явно какое-нибудь другое число).

`show listsize`
: Отобразить количество строк, выводимых по команде `list`.

Повторение команды `list` нажатием ⟨RET⟩ отбрасывает аргумент, так что это эквивалентно вводу просто `list`. Это полезнее, чем вывод тех же самых строк снова. Исключение сделано для параметра '`-`'; этот параметр сохраняется при повторе команды, так что каждое повторение приводит к перемещению вверх по исходному файлу.

Обычно команда `list` ожидает от вас ноль, один или два *указателя строк*. Указатели строк определяют строки исходного текста; существует несколько способов их задания, но результат всегда заключается в задании строки исходного текста. Вот полное описание возможных параметров команды `list`:

`list` *указ-стр*
: Вывести строки, расположенные вокруг строки, определяемой *указ-стр*.

`list` *перв*`,`*посл*
: Вывести строки с *перв* до *посл*. Оба параметра являются указателями строк.

`list ,`*посл*
: Вывести строки, расположенные перед *посл*.

`list` *перв*`,`
: Вывести строки, начиная с *перв*.

`list +` Вывести строки, расположенные сразу за последними выведенными.

`list -` Вывести строки, расположенные непосредственно перед последними выведенными.

`list` Описано в предыдущей таблице.

Ниже перечислены способы указания одиночной строки исходного текста—все виды указателей строк.

номер Определяет строку с номером *номер* из текущего исходного файла. Если в качестве параметров к команде `list` задано два указателя строк, это относится к тому же исходному файлу, что и первый указатель строки.

`+`*смещение*
: Указывает на строку, смещенную вперед на *смещение* строк относительно последней выведенной строки. Когда используется в качестве второго указателя строки для команды `list`, имеющей два указателя, задает строку, смещенную на *смещение* строк вниз относительно строки, определенной первым указателем.

`-`*смещение*
: Указывает на строку, расположенную на *смещение* строк раньше последней выведенной строки.

имя-файла`:`*номер*
: Задает строку *номер* из исходного файла *имя-файла*.

функция Определяет строку, с которой начинается тело функции *функция*. Например, в Си это строка с открывающейся фигурной скобкой.

имя-файла`:`*функция*
: Определяет строку с открывающейся фигурной скобкой, с которой начинается тело функции *функция* в файле *имя-файла*. Имя файла необходимо лишь для того, чтобы избежать неоднозначности, когда в различных исходных файлах есть одинаково названные функции.

`*`*адрес* Определяет строку, соответствующую адресу *адрес* программы. *адрес* может быть любым выражением.

7.2 Поиск в исходных файлах

Существуют две команды для поиска по регулярному выражению в текущем исходном файле.

`forward-search` *рег-выраж*
`search` *рег-выраж*
: Команда ‘`forward-search` *рег-выраж*’ проверяет на соответствие регулярному выражению *рег-выраж* каждую строку, начиная со строки, следующей за последней выведенной. Найденная строка выводится. Вы можете использовать синоним ‘`search` *рег-выраж*’ или сокращать имя команды как `fo`.

`reverse-search` *рег-выраж*
: Команда '`reverse-search` *рег-выраж*', двигаясь назад, проверяет на соответствие регулярному выражению *рег-выраж* каждую строку, начиная с предшествующей последней выведенной. Найденная строка выводится. Вы можете сокращать эту команду как `rev`.

7.3 Определение каталогов с исходными файлами

Исполняемые программы иногда не сохраняют имена каталогов, в которых находились исходные файлы, из которых они скомпилированы, а хранят лишь имена файлов. Даже если они их сохранили, каталоги могли быть перемещены в период между компиляцией и сеансом отладки. У GDB есть список каталогов для поиска исходных файлов; он называется *путь для исходных файлов*. Каждый раз, когда GDB требуется исходный файл, он перебирает по порядку все каталоги из этого списка, пока не находит файл с требуемым именем. Заметьте, что пути поиска исполняемых файлов для этой цели *не* используются, как не используется и текущий рабочий каталог, если только он не присутствует в пути для исходных файлов.

Если GDB не может найти исходный файл, используя путь для исходных файлов, а в объектном файле программы указан какой-либо каталог, GDB просматривает также и его. В последнюю очередь, если путь для исходных файлов пуст и запись о каталоге компиляции отсутствует, GDB просматривает текущий каталог.

При переустановке или переупорядочивании пути для исходных файлов, GDB очищает любую запомненную им информацию о том, где исходные файлы были найдены и о расположении строк в них.

Когда вы вызываете GDB, путь для исходных файлов содержит только '`cdir`' и '`cwd`', в этом порядке. Для добавления других каталогов, используйте команду `directory`.

`directory` *имя-каталога* `...`

`dir` *имя-каталога* `...`
: Добавить каталог *имя-каталога* в начало пути для исходных файлов. Этой команде могут быть заданы несколько имен, разделенные ':' (';' в MS-DOS и MS-Windows, где ':' обычно является частью абсолютного имени файла) или пробелом. Вы можете указать каталог, который уже содержится в пути для исходных файлов; это переместит его в начало, так что GDB будет просматривать его раньше.

  Вы можете использовать строку '`$cdir`' для ссылки на каталог компиляции (если информация о нем сохранена), и '`$cwd`' для ссылки на текущий рабочий каталог. '`$cwd`' не есть то же самое, что '`.`'. Первая отслеживает текущий рабочий каталог, который может меняться во время вашего сеанса работы с GDB, тогда как вторая сразу преобразовывается в текущий каталог в момент его добавления в путь для исходных файлов.

`directory`
: Очистить путь для файлов с исходными текстами. Эта команда требует подтверждения.

`show directories`
: Вывести путь поиска исходных файлов: показать, какие каталоги он содержит.

Если ваш путь для исходных файлов перемешан с уже неиспользуемыми каталогами, GDB может иногда вызвать недоумение, найдя неправильный вариант исходного файла. Вы можете исправить ситуацию следующим образом:

1. Использовать `directory` без параметров, чтобы очистить путь поиска исходных файлов.

2. Использовать `directory` с подходящими аргументами, чтобы переустановить каталоги, которые вы хотите видеть в пути для исходных файлов. Вы можете добавить все каталоги одной командой.

7.4 Исходный текст и машинный код

Вы можете использовать команду `info line`, чтобы отобразить строки исходного текста в программные адреса (и наоборот), и команду `disassemble`, чтобы вывести диапазон адресов в виде машинных инструкций. При запуске в режиме GNU Emacs, команда `info line` выводит стрелку, указывающую на заданную строку. Также `info line` выводит адреса как в символьной форме, так и в шестнадцатеричной.

`info line` *указ-стр*
: Выводит начальный и конечный адреса скомпилированного кода, соответствующего строке исходного текста *указ-стр*. Вы можете определить строки исходного текста любым способом, воспринимаемым командой `list` (см. Раздел 7.1 [Вывод строк исходного текста], с. 55).

Например, мы можем использовать `info line` для определения положения объектного кода первой строки функции `m4_changequote`:

```
(gdb) info line m4_changequote
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

Мы также можем запросить (используя **адрес* как форму задания *указ-стр*), какая строка исходного текста соответствует определенному адресу:

```
(gdb) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.
```

После `info line`, адрес, используемый по умолчанию для команды `x`, меняется на начальный адрес строки, так что ‘`x/i`’ достаточно для начала исследования машинного кода (см. Раздел 8.5 [Исследование памяти], с. 65). Этот адрес также сохраняется как значение вспомогательной переменной `$_` (см. Раздел 8.9 [Вспомогательные переменные], с. 73).

`disassemble`
: Эта специализированная команда служит для дампа диапазона памяти в виде машинных инструкций. Диапазоном памяти по умолчанию является функция, в которой находится счетчик программы в выбранном кадре. Одиночным параметром этой команды является значение счетчика программы; GDB выводит дамп функции, которой принадлежит указанный адрес. Два параметра определяют диапазон адресов для дампа (первый включается, второй исключается).

Следующий пример показывает результат дисассемблирования диапазона адресов кода HP PA-RISC 2.0:

```
(gdb) disas 0x32c4 0x32e4
Dump of assembler code from 0x32c4 to 0x32e4:
0x32c4 <main+204>:      addil 0,dp
0x32c8 <main+208>:      ldw 0x22c(sr0,r1),r26
0x32cc <main+212>:      ldil 0x3000,r31
0x32d0 <main+216>:      ble 0x3f8(sr4,r31)
0x32d4 <main+220>:      ldo 0(r31),rp
0x32d8 <main+224>:      addil -0x800,dp
0x32dc <main+228>:      ldo 0x588(r1),r26
0x32e0 <main+232>:      ldil 0x3000,r31
End of assembler dump.
```

Некоторые архитектуры имеют несколько широко используемых наборов мнемоник инструкций или другой синтаксис.

`set disassembly-flavor` *набор-инструкций*

Выбрать набор инструкций для использования при дисассемблировании программы командами `disassemble` и `x/i`.

В настоящее время, эта команда определена только для Intel x86. Вы можете установить *набор-инструкций* в `intel` или `att`. По умолчанию установлено `att`, диалект AT&T используется по умолчанию ассемблерами Unix на архитектурах, базирующихся на x86.

8 Исследование данных

Для исследования данных в вашей программе обычно используется команда `print` (сокращенно `p`) или ее синоним `inspect`. Она вычисляет и выводит значение выражения, записанного на том же языке, что и ваша программа (см. Глава 9 [Использование GDB с различными языками], с. 77).

`print` *выраж*
`print /`*f* *выраж*
: *выраж* является выражением (на исходном языке). По умолчанию, значение *выраж* выводится в формате, соответствующем его типу данных; вы можете выбрать другой формат, указав '`/`*f*', где *f*—буква, определяющая формат; смотрите Раздел 8.4 [Форматы вывода], с. 64.

`print`
`print /`*f*
: Если вы опустите *выраж*, GDB отображает последнее значение снова (из *истории значений*; см. Раздел 8.8 [История значений], с. 72). Это предоставляет вам удобный способ изучить то же самое значение в другом формате.

Команда `x` позволяет исследовать данные на более низком уровне. Она исследует данные в памяти по указанному адресу и выводит их в указанном формате. См. Раздел 8.5 [Исследование памяти], с. 65.

Если вас интересует информация о типах или о том, как объявлены поля структуры или класса, используйте команду `ptype` *выраж* вместо `print`. См. Глава 10 [Исследование таблицы символов], с. 97.

8.1 Выражения

`print` и многие другие команды GDB допускают в качестве параметра выражение и вычисляют его значение. В выражении GDB допустимо использование любого типа констант, переменных или операторов, определенных в используемом вами языке программирования, включая условные выражения, вызовы функций, приведение типов и строковые постоянные. К сожалению, исключением являются символы, определенные командами препроцессора `#define`.

GDB поддерживает константы-массивы в выражениях, введенных пользователем. Синтаксис следующий: `{`*элемент*, *элемент*`...}`. Например, вы можете использовать команду `print {1, 2, 3}`, чтобы создать в памяти массив, который будет доступен в программе так же, как выделенный функцией `malloc`.

По причине широкого распространения Си, большинство выражений в примерах этого руководства написаны на Си. См. Глава 9 [Использование GDB с различными языками], с. 77, для информации об использовании выражений в других языках.

В этом разделе мы обсуждаем операторы, которые вы можете использовать в выражениях GDB независимо от используемого вами языка программирования.

Приведения типов поддерживается во всех языках, а не только в Си, так как бывает очень полезно преобразовать число в указатель, чтобы исследовать структуру, расположенную по этому адресу в памяти.

GDB поддерживает эти операторы, в дополнении к следующим, являющимися общими для языков программирования:

`@`
: '`@`' является бинарным оператором, позволяющим рассматривать области памяти как массивы. См. Раздел 8.3 [Исскуственные массивы], с. 63, для дополнительной информации.

`::`
: '`::`' позволяет вам указывать переменную в терминах файла или функции, где она определена. См. Раздел 8.2 [Переменные программы], с. 62.

{*тип*} *адрес*
: Ссылается на объект типа *тип*, хранящийся в памяти по адресу *адрес*. *Адрес* может быть любым выражением, значением которого является целое число или указатель (но вокруг бинарных операторов, также как и вокруг оператора приведения типа, требуются скобки). Эта конструкция допустима, независимо от того, какого типа данные предположительно расположены по *адресу*.

8.2 Переменные программы

Чаще всего в качестве выражения используется имя переменной вашей программы.

Переменные в выражениях трактуются в контексте выбранного кадра стека (см. Раздел 6.3 [Выбор кадра стека], с. 52); они могут быть либо

- глобальными (или статическими)

либо

- видимыми из точки выполнения в данном кадре, в соответствии с правилами определения области видимости языка программирования.

Это означает, что в функции

```
foo (a)
     int a;
{
  bar (a);
  {
    int b = test ();
    bar (b);
  }
}
```

вы можете исследовать и использовать переменную `a` всякий раз, когда ваша программа выполняется в пределах функции `foo`, но вы можете использовать или исследовать переменную `b` только тогда, когда ваша программа выполняется внутри блока, в котором она объявлена.

Есть исключение: вы можете ссылаться на переменную или функцию, областью видимости которой является единственный исходный файл, даже если точка текущего выполнения в нем не находится. Допускается существование нескольких переменных или функций с одинаковым именем (в различных исходных файлах). Если это так, обращение к этому имени приводит к непредсказуемым результатам. Если хотите, вы можете указать статическую переменную в конкретной функции или в файле, используя двойное двоеточие:

файл`::`*переменная*
функция`::`*переменная*

Здесь *файл* или *функция*—название контекста для статической *переменной*. В первом случае вы можете использовать кавычки, чтобы GDB рассматривал имя файла как одно слово; например, чтобы вывести глобальное значение переменной `x`, определенной в '`f2.c`':

```
(gdb) p 'f2.c'::x
```

Такое использование '`::`' крайне редко конфликтует с похожим использованием той же записи в Си++. GDB также поддерживает использование оператора определения области видимости Си++ в выражениях.

> *Предупреждение:* В некоторых случаях, в определенной точке функции (сразу после входа в новую область видимости, и непосредственно перед выходом из нее) может показаться, что локальная переменная имеет неверное значение.

Вы можете столкнуться с этой проблемой при пошаговом выполнении по одной машинной инструкции. Она возникает из-за того, что на большинстве машин процедура установки кадра стека (включая определения локальных переменных) занимает более одной инструкции; если вы производите пошаговое выполнение по одной машинной инструкции, может показаться, что переменная имеет неверное значение, пока кадр стека не будет полностью построен. При выходе, для уничтожения кадра стека обычно также требуется более одной инструкции; после начала пошагового выполнения этой группы инструкций, определения локальных переменных могут пропасть.

Это также может случиться, когда компилятор делает значительную оптимизацию. Чтобы быть уверенным, что вы всегда видите точные значения, отключите всю оптимизацию при компиляции.

Другой возможный эффект оптимизации компилятора заключается в уничтожении неиспользуемых переменных, или в присвоении переменных регистрам (а не адресам памяти). В зависимости от поддержки таких ситуаций, предоставляемой форматом отладочной информации, который использует компилятор, GDB может не вывести значения таких локальных переменных. Если это происходит, GDB выведет сообщение, подобное этому:

```
No symbol "foo" in current context.
```

Для решения таких проблем, либо перекомпилируйте программу без оптимизации, или используйте другой формат отладочной информации, если компилятор поддерживает несколько таких форматов. Например GCC, компилятор GNU Си/Си++, обычно поддерживает ключ ‘`-gstabs`’. ‘`-gstabs`’ создает отладочную информацию в формате, являющимся развитием таких форматов, как COFF. У вас может быть возможность использовать DWARF-2 (‘`-gdwarf-2`’), который тоже является эффективной формой представления отладочной информации. Смотрите раздел “Опции для отладки вашей программы или GNU CC” в *Использование GNU CC*, для дополнительной информации.

8.3 Искусственные массивы

Часто бывает полезным вывести несколько объектов одного типа, расположенных в памяти последовательно; часть массива или динамический массив, для которого в программе существует только указатель.

Вы можете это сделать, обращаясь к непрерывному участку памяти как к *искусственному массиву*, используя бинарный оператор ‘`@`’. Левым операндом для ‘`@`’ должен быть первый элемент желаемого массива, и он должен быть индивидуальным объектом. Правым операндом должна быть длина массива. Результатом операции будет массив, все элементы которого имеют тот же тип, что и левый аргумент. Первым элементом массива является левый аргумент; второй элемент формируется из байтов памяти, непосредственно следующих за байтами, содержащими первый элемент, и так далее. Например, если в программе есть строка

```
int *array = (int *) malloc (len * sizeof (int));
```

то вы можете вывести содержимое `array` с помощью

```
p *array@len
```

Левый операнд операции ‘`@`’ должен находиться в памяти. Значения массивов, полученных операцией ‘`@`’, при индескации ведут себя точно так же, как и другие массивы, и приводятся к указателям при использовании в выражениях. Искусственные массивы чаще всего появляются в выражениях через историю значений (см. Раздел 8.8 [История значений], с. 72), после вывода одного из них.

Другой способ создания искусственного массива—использование приведения типов. Оно заново интерпретирует значение так, как если бы оно было массивом. Значение не обязано находиться в памяти.

```
(gdb) p/x (short[2])0x12345678
$1 = {0x1234, 0x5678}
```

Если вы опускаете длину массива (как в '`(`*тип*`[])`*значение*'), GDB для удобства вычисляет его размер для заполнения значениями (как '`sizeof(`*значение*`)/sizeof(`*тип*`)`':

```
(gdb) p/x (short[])0x12345678
$2 = {0x1234, 0x5678}
```

Иногда механизма искусственных массивов бывает недостаточно; в сравнительно сложных структурах данных, интересующие нас элементы могут не быть смежными—например, если вас интересуют значения указателей в массиве. Одно из полезных решений этой проблемы—использование вспомогательной переменной (см. Раздел 8.9 [Вспомогательные переменные], с. 73) в качестве счетчика в выражении, выводящем первое интересующее нас значение, а затем повторять это выражение нажатием ⟨RET⟩. Предположим, например, у вас есть массив `dtab` указателей на структуры, и вас интересуют значения полей `fv` в каждой структуре. Ниже приведен пример ваших возможных действий:

```
set $i = 0
p dtab[$i++]->fv
⟨RET⟩
⟨RET⟩
...
```

8.4 Форматы вывода

По умолчанию, GDB печатает значение в соответствии с его типом. Это не всегда отвечает вашему желанию. Например, вы можете захотеть вывести число в шестнадцатеричной записи, или указатель в десятичной. Или вы можете захотеть просмотреть данные по некоторому адресу в памяти в виде строки символов или в виде инструкций. Для этого, при выводе значения укажите *формат вывода*.

Простейшим применением форматов вывода является форматирование вывода уже вычисленного выражения. Это осуществляется путем начала параметров команды `print` с косой черты и символа формата. Поддерживаются следующие символы формата:

`x` Рассматривать биты значения как целое, и вывести целое в шестнадцатеричном виде.

`d` Вывести как десятичное целое со знаком.

`u` Вывести как десятичное целое без знака.

`o` Вывести как восьмеричное целое.

`t` Вывести как целое в двоичном виде. Буква '`t`' означает "two".[1]

`a` Вывести в виде адреса, как абсолютного в шестнадцатеричной записи, так и в виде смещения от ближайшего предшествующего символа. Вы можете использовать этот формат для того, чтобы определить, где (в какой функции) расположен какой-либо неизвестный адрес:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

`c` Рассматривать как целое и вывести в виде строковой постоянной.

`f` Рассматривать биты значения как число с плавающей точкой и вывести с использованием обычного синтаксиса для чисел с плавающей точкой.

[1] '`b`' не может быть использовано, потому что эти символы формата также используются с командой `x`, где '`b`' означает "byte"; смотрите Раздел 8.5 [Исследование памяти], с. 65.

Например, чтобы вывести счетчик программы в шестнадцатеричном виде (см. Раздел 8.10 [Регистры], с. 74), введите

```
p/x $pc
```

Обратите внимание, что перед косой чертой не требуется пробела, потому что имена команд в GDB не могут содержать косую черту.

Чтобы вывести последнее значение из истории значений в другом формате, вы можете воспользоваться командой `print` лишь с указанием формата и без выражения. Например, ‘`p/x`’ выведет последнее значение в шестнадцатеричной форме.

8.5 Исследование памяти

Вы можете использовать команду `x` (от слова “examine”) для исследования памяти в одном из нескольких форматов, независимо от типов данных вашей программы.

`x`/*nfu адрес*
`x` *адрес*
`x` Для исследования памяти используйте команду `x`.

n, *f* и *u*—необязательные параметры, определяющие, сколько памяти отобразить и в каком формате; *адрес*—это выражение, задающее адрес, с которого вы хотите начать отображение памяти. Если вы используете значения по умолчанию для *nfu*, то вам не нужно вводить косую черту ‘`/`’. Некоторые команды устанавливают удобные значения по умолчанию для *адреса*.

n, счетчик повторений
: Счетчик повторений является десятичным целым числом; по умолчанию 1. Он определяет, сколько памяти отобразить (считая в единицах *u*).

f, формат отображения
: Формат отображения—это один из форматов, используемых командой `print`, ‘`s`’ (строка, оканчивающаяся нулем), или ‘`i`’ (машинная инструкция). Первоначально, значением по умолчанию установлено ‘`x`’ (шестнадцатеричная форма). Значение по умолчанию изменяется каждый раз, когда вы используете либо `x`, либо `print`.

u, размер единицы измерений
: Размер единицы измерений может быть одним из

  `b` Байты.

  `h` Полуслова (два байта).

  `w` Слова (четыре байта). Это первоначальное значение по умолчанию.

  `g` Длинные слова (восемь байт).

  Каждый раз, когда вы определяете размер единицы измерений командой `x`, этот размер становится размером по умолчанию при последующем использовании `x`. (Для форматов ‘`s`’ и ‘`i`’, размер единицы измерений игнорируется и обычно не пишется.)

адрес, начальный адрес отображения
: *адрес*—это адрес, с которого вы хотите, чтобы GDB начинал отображение памяти. Значение выражения не обязано должно быть указателем (хотя может им быть); оно всегда интерпретируется как целый адрес байта в памяти. См. Раздел 8.1 [Выражения], с. 61, для дополнительной информации о выражениях. Значением по умолчанию для *адреса* обычно является адрес, следующий за

последним изученным адресом, но некоторые другие команды также устанавливают это значение: `info breakpoints` (в адрес последней выведенной точки останова), `info line` (в начальный адрес строки) и `print` (если вы используете эту команду для отображения значения из памяти).

Например, '`x/3uh 0x54320`'—запрос на вывод трех полуслов (`h`) памяти в формате беззнаковых десятичных целых ('`u`'), начиная с адреса `0x54320`. '`x/4xw $sp`' выводит четыре слова ('`w`') памяти, расположенные над указателем стека (здесь '`$sp`'; см. Раздел 8.10 [Регистры], с. 74), в шестнадцатеричном виде ('`x`').

Так как все буквы, обозначающие размер единиц измерения, отличаются от букв, определяющих форматы вывода, вы не должны запоминать, формат или размер единиц измерений указывается раньше; это можно делать в любом порядке. Спецификации вывода '`4xw`' и '`4wx`' означают в точности одно и то же. (Однако, число *n* должно быть первым; '`wx4`' не сработает.)

Хотя размер единицы измерения *u* игнорируется для форматов '`s`' и '`i`', тем не менее вы можете воспользоваться счетчиком повторений *n*; например, '`3i`' указывает, что вы хотите вывести три машинные инструкции, включая любые операнды. Команда `disassemble` предоставляет альтернативный способ исследования машинных инструкций; смотрите Раздел 7.4 [Исходный и машинный код], с. 58.

Все значения по умолчанию для аргументов команды `x` разработаны таким образом, чтобы облегчить продолжение сканирования памяти с минимальными конкретизациями при очередном использовании `x`. Например, после того, как вы просмотрели три машинные инструкции с помощью '`x/3i` *адрес*', вы можете просмотреть следующие семь, используя просто '`x/7`'. Если вы повторяете команду `x` нажатием ⟨RET⟩, число повторений *n* остается прежним; другие параметры берутся по умолчанию, как для последовательных использований `x`.

Адреса и их содержимое, выводимые командой `x`, не сохраняются в истории значений, так как они мешали бы. Вместо этого, GDB делает их доступными для последующего использования в выражениях как значения вспомогательных переменных `$_` и `$__`. После команды `x`, последний исследованный адрес доступен для использования в выражениях во вспомогательной переменной `$_`. Содержимое этого адреса, исследованное только что, доступно во вспомогательной переменной `$__`.

Если команде `x` задан счетчик повторений, адрес и его содержимое сохраняются из последнего выведенного элемента памяти; это не то же самое, что последний выведенный адрес, если в последней строке вывода были отображены несколько элементов.

8.6 Автоматическое отображение

Если вам необходимо часто выводить значение какого-либо выражения (чтобы увидеть, как оно меняется), вы можете добавить его в *список автоматического отображения*, чтобы GDB выводил его значение каждый раз при остановке вашей программы. Каждому выражению, добавленному в список, присваивается идентификационный номер; чтобы удалить выражение из списка, вы указываете этот номер. Автоматическое отображение выглядит следующим образом:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

Это отображение показывает номера элементов, выражения и их текущие значения. Как и при отображении, запрашиваемом вручную с помощью `x` или `print`, вы можете указать предпочитаемый формат вывода; фактически, `display` определяет, следует использовать `print` или `x`, в зависимости от того, на сколько жесткая ваша спецификация формата: используется `x`, если вы указываете размер элемента или один из двух форматов ('`i`' и `s`), которые поддерживаются только `x`; в остальных случаях используется `print`.

`display` *выраж*
: Добавляет выражение *выраж* к списку выражений, отображаемых каждый раз, когда ваша программа останавливается. См. Раздел 8.1 [Выражения], с. 61.

  `display` не повторяется, если вы повторно нажимаете ⟨RET⟩ после ее использования.

`display/`*формат выраж*
: Если *формат* определяет только формат вывода, а не размер или счетчик повторений, выражение *выраж* добавляется в список автоматического отображения, но его отображение осуществляется в указанном формате *формат*. См. Раздел 8.4 [Форматы вывода], с. 64.

`display/`*формат адрес*
: Если *форматом* является ‘`i`’ или ‘`s`’, или он включает в себя размер элементов или их число, выражение *адрес* добавляется как адрес памяти для исследования при каждой остановке вашей программы. Под исследованием в данном случае подразумевается выполнение ‘`x/`*формат адрес*’. См. Раздел 8.5 [Исследование памяти], с. 65.

Например, команда ‘`display/i $pc`’ может быть полезна, чтобы при каждой остановке видеть машинную инструкцию, которая будет выполняться следующей (‘`$pc`’—это общее обозначение счетчика программы; см. Раздел 8.10 [Регистры], с. 74).

`undisplay` *номера...*
`delete display` *номера...*
: Удалить элементы с номерами *номера* из списка выражений, подлежащих отображению.

  `undisplay` не повторяется при последующем нажатии ⟨RET⟩. (Иначе вы сразу получили бы сообщение об ошибке ‘`No display number ...`’.)

`disable display` *номера...*
: Отключить отображение элементов с номерами *номера*. Отключенные элементы не выводятся автоматически, но и не забываются. Впоследствии их можно снова включить.

`enable display` *номера...*
: Включить отображение элементов с номерами *номера*. Выражения, соответствующие этим номерам, снова будут выводиться автоматически, пока вы укажете обратное.

`display`
: Отобразить текущие значения выражений из списка, точно так же, как это происходит при остановке вашей программы.

`info display`
: Вывести список выражений, ранее установленных для автоматического отображения, каждое с его номером элемента, но не показывая значений. Список включает отключенные выражения, с соответствующей пометкой. Он также включает в себя выражения, которые не могут быть показаны прямо сейчас, потому что обращаются к автоматическим переменным, недоступным в данный момент.

Если отображаемое выражение обращается к локальным переменным, оно не имеет смысла вне того лексического контекста, для которого оно устанавливалось. Такое выражения отключается, как только выполнение входит в контекст, где одна из его переменных становится неопределенной. Например, если вы дадите команду `display last_char`, находясь внутри функции с аргументом `last_char`, GDB будет отображать этот аргумент, пока программа останавливается внутри этой функции. Как только она остановится где-то еще—где нет переменной `last_char`—отображение будет отключено автоматически.

Вы можете снова включить его при следующей остановке программы там, где `last_char` будет вновь иметь смысл.

8.7 Параметры вывода

GDB предоставляет следующие способы управления выводом массивов, структур и символов.

Данные параметры полезны при отладке программ на любом языке:

`set print address`
`set print address on`
: GDB выводит адреса памяти, показывающие положение стека, структур, указателей, точек останова, и так далее, даже когда он отображает также содержимое этих адресов. Значение по умолчанию установлено в `on`. Например, вот как выглядит отображение кадра стека с установленным `set print address on`:

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530         if (lquote != def_lquote)
```

`set print address off`
: Не выводить адреса при отображении их содержимого. Вот, например, тот же кадр стека, отображенный с установкой `set print address off`:

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530         if (lquote != def_lquote)
```

Вы можете использовать ‘`set print address off`’, чтобы удалить все машинно-зависимые отображения из интерфейса GDB. Например, с `print address off`, вы должны получить одинаковый текст для цепочек вызовов на всех машинах, независимо от того, включают они указатели в качестве аргументов или нет.

`show print address`
: Показать, должны выводиться адреса или нет.

При выводе адреса в символьной форме, GDB обычно выводит ближайший предшествующий символ плюс смещение. Если этот символ не определяет адрес однозначно (например, это имя, областью действия которого является один исходный файл), вам может потребоваться дать пояснения. Один из способов это сделать—с помощью `info line`; например, ‘`info line *0x4537`’. Альтернативный способ заключается в том, чтобы GDB выводил имя исходного файла и номер строки при выводе символьного адреса:

`set print symbol-filename on`
: Велит GDB выводить имя исходного файла и номер строки символа в символьной форме адреса.

`set print symbol-filename off`
: Не выводить имя исходного файла и номер строки символа. Принимается по умолчанию.

`show print symbol-filename`
: Показать, будет GDB выводить имя исходного файла и номер строки в символьной форме адреса или нет.

Другая ситуация, в которой полезно показывать имена файлов и номера строк, возникает при дисассемблировании кода; GDB показывает вам номер строки и исходный файл, которые соответствуют каждой инструкции.

Вы также можете захотеть видеть символьную форму только в том случае, если выводимый адрес достаточно близок к ближайшему предшествующему символу:

`set print max-symbolic-offset` *макс-смещение*
: Велит GDB выводить символьные формы только тех адресов, для которых смещение между ближайшим предшествующим символом и адресом меньше, чем *макс-смещение*. По умолчанию значение *макс-смещение* равно 0; в этом случае GDB всегда выводит адрес в символьной форме, если ему предшествует хоть какой-нибудь символ.

`show print max-symbolic-offset`
: Запрашивает информацию о максимальном смещении, для которого GDB выводит символьную форму адреса.

Если у вас есть указатель, и вы не знаете, на что он указывает, попробуйте '`set print symbol-filename on`'. Затем вы можете определить название и исходный файл переменной, на которую он указывает, используя '`p/a` *указатель*'. Это интерпретирует адрес в символьной форме. Например, здесь GDB показывает, что переменная `ptt` указывает на другую переменную `t`, определенную в файле '`hi2.c`':

```
(gdb) set print symbol-filename on
(gdb) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

Предупреждение: Для указателей, указывающих на локальные переменные, '`p/a`' не показывает символьное имя и имя файла, которому принадлежит объект ссылки, даже если установлен соответствующий параметр `set print`.

Другие установки управляют выводом объектов различных типов:

`set print array`
`set print array on`
: Структурный вывод массивов. Этот формат удобенее для чтения, но занимает больше места. По умолчанию отключено.

`set print array off`
: Вернуться к сжатому формату вывода массивов.

`show print array`
: Показать, какой формат (сжатый или структурный) выбран для отображения массивов.

`set print elements` *число-элементов*
: Установить ограничение на количество выводимых GDB элементов массива. Если GDB выводит большой массив, вывод прерывается после того, как будет выведено установленное командой `set print elements` число элементов. Это ограничение также действует при отображении строк. Когда GDB стартует, этот предел принимается равным 200. Установка *число-элементов* в ноль означает, что вывод не ограничен.

`show print elements`
: Показать количество элементов большого массива, которые будут выведены GDB. Если это число равно 0, вывод не ограничивается.

`set print null-stop`
: Указывает GDB прекращать вывод символов массива, как только встретится первый NULL. Это полезно, когда большие массивы фактически содержат только короткие строки. По умолчанию отключено.

`set print pretty on`
: Велит GDB выводить структуры в формате с отступами, по одному элементу в строке, например:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

`set print pretty off`
: Указывает GDB выводить структуры в компактном формате, как здесь:

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, \
meat = 0x54 "Pork"}
```

Этот формат устанавливается по умолчанию.

`show print pretty`
: Показать, какой формат GDB использует для вывода структур.

`set print sevenbit-strings on`
: Осуществлять вывод, используя только семибитные символы; если этот параметр установлен, GDB отображает любые восьмибитные символы (в строках или символьных значениях), используя запись *nnn*. Эта установка очень удобна, если вы работаете на английском (ASCII) и используете старший бит символов как маркер или “мета”-бит.

`set print sevenbit-strings off`
: Выводить восьмибитные символы полностью. Это позволяет использовать большее количество международных наборов символов, и устанавливается по умолчанию.

`show print sevenbit-strings`
: Показать, выводит GDB только семибитные литеры или нет.

`set print union on`
: Велит GDB выводить объединения, содержащиеся в структурах. Устанавливается по умолчанию.

`set print union off`
: Указывает GDB не выводить объединения, содержащиеся в структурах.

`show print union`
: Запросить GDB, будет ли он выводить объединения, содержащиеся в структурах.

Например, пусть даны описания

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
              Bug_forms;
```

```
struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};

struct thing foo = {Tree, {Acorn}};
```

с установленным `set print union on`, команда ‘`p foo`’ выведет

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

а с установленным `set print union off`, эта же команда выведет

```
$1 = {it = Tree, form = {...}}
```

Следующие установки представляют интерес при отладке программ на Си++:

`set print demangle`
`set print demangle on`
: Печатать идентификаторы Си++ в их изначальной, а не в закодированной (“искаженной”) форме, передаваемой ассемблеру и компоновщику для сборки с контролем типа. Установлено по умолчанию.

`show print demangle`
: Показать, в искаженной или восстановленной форме выводятся идентификаторы Си++.

`set print asm-demangle`
`set print asm-demangle on`
: Выводить идентификаторы Си++ в их исходной форме, а не в искаженной, даже при выводе ассемблерного кода, например при дисассемблировании инструкций. По умолчанию отключено.

`show print asm-demangle`
: Показать, в закодированной или восстановленной форме выводятся имена Си++ при выводе кода ассемблера.

`set demangle-style` *стиль*
: Выбрать одну из нескольких схем кодирования, используемых различными компиляторами для представления имен Си++. Параметр *стиль* может быть следующим:

  `auto` — Позволить GDB выбрать стиль декодирования посредством изучения вашей программы.

  `gnu` — Декодирование основывается на алгоритме кодирования компилятора GNU Си++ (`g++`). Устанавливается по умолчанию.

  `hp` — Декодирование основывается на алгоритме кодирования HP ANSI Си++ (`aCC`).

  `lucid` — Декодирование основывается на алгоритме кодирования компилятора Lucid Си++ (`lcc`).

  `arm` — Декодировать, используя алгоритм из *C++ Annotated Reference Manual*. **Предупреждение:** одной этой установки недостаточно, чтобы производить отладку исполняемых программ, сгенерированных `cfront`. Чтобы реализовать это, GDB требует дальнейших усовершенствований.

Если вы опустите *стиль*, то увидите список возможных форматов.

`show demangle-style`
: Отобразить текущий стиль кодирования, используемый для декодирования символов Си++.

`set print object`
`set print object on`
: При отображении указателя на объект, идентифицировать *фактический* (производный), а не *объявленный* тип объекта, используя таблицу виртуальных функций.

`set print object off`
: Отображать только объявленный тип объекта, не ссылаясь на таблицу виртуальных функций. Устанавливается по умолчанию.

`show print object`
: Показать, какой из типов объекта выводится.

`set print static-members`
`set print static-members on`
: Выводить статические члены при отображении объекта Си++. Установлено по умолчанию.

`set print static-members off`
: Не выводить статические члены при отображении объекта Си++.

`show print static-members`
: Показать, выводятся статические члены Си++ или нет.

`set print vtbl`
`set print vtbl on`
: Осуществлять структурный вывод таблиц виртуальных функций Си++. По умолчанию отключено. (Команды `vtbl` не работают для программ, скомпилированных компилятором HP ANSI Си++ (`aCC`).)

`set print vtbl off`
: Не производить структурного вывода таблиц виртуальных функций Си++.

`show print vtbl`
: Показать, производится структурный вывод таблиц виртуальных функций Си++ или нет.

8.8 История значений

Значения, выведенные командой `print`, сохраняются в *истории значений* GDB. Это позволяет вам обращаться к ним в других выражениях. Значения сохраняются, пока таблица символов не будет заново считана или уничтожена (например, командами `file` или `symbol-file`). При изменении таблицы символов, история значений уничтожается, так как значения могут содержать указатели на типы, определенные в таблице символов.

Выведенным значениям присваиваются *номера в истории*, по которым вы можете на них ссылаться. Эти номера являются последовательными целыми числами, начинающимися с единицы. Команда `print` показывает номер в истории, присвоенный значению, выводя перед ним '`$`*номер* `=` ', где *номер*—это номер в истории.

Для обращения к какому-либо предшествующему значению, используйте '`$`', за которым следует номер в истории. Способ, которым `print` маркирует вывод продуман так, чтобы напоминать вам об этом. Просто `$` ссылается на самое последнее значение в истории, а `$$`—на предпоследнее. `$$`*n* ссылается на *n*-е с конца значение; `$$2`—значение, находящееся перед `$$`, `$$1` эквивалентно `$$`, а `$$0` эквивалентно `$`.

Предположим, например, вы только что вывели указатель на структуру и хотите посмотреть ее содержимое. Для этого достаточно ввести

```
p *$
```

Если у вас есть цепочка структур, где компонента `next` указывает на следующую, вы можете вывести содержимое следующей структуры так:

```
p *$.next
```

Вы можете выводить последовательные звенья в цепочке повторяя эту команду. Это можно сделать простым нажатием ⟨RET⟩.

Обратите внимание, что в историю записываются значения, а не выражения. Если значение `x` равно 4, и вы наберете:

```
print x
set x=5
```

то значение, записанное в историю значений командой `print`, будет по-прежнему равно 4, хотя значение `x` изменилось.

`show values`
: Вывести из истории последние десять значений, с их номерами. Это похоже на команду '`p $$9`', повторенную десять раз, за исключением того, что `show values` не изменяет историю.

`show values` *n*
: Вывести десять значений из истории, расположенных вокруг элемента с номером *n*.

`show values +`
: Вывести десять значений из истории, следующих сразу после последнего выведенного значения. Если доступных значений больше нет, `show values +` не выводит ничего.

Нажатие ⟨RET⟩ для повтора `show values` *n* действует точно так же, как '`show values +`'.

8.9 Вспомогательные переменные

GDB предоставляет *вспомогательные переменные*, которые вы можете в нем использовать, чтобы сохранить значение и обратиться к нему позже. Эти переменные существуют только в GDB; они не являются частью вашей программы и установка вспомогательной переменной не оказывает непосредственного влияния на ее дальшейшее выполнение. Поэтому вы можете пользоваться ими совершенно свободно.

Имена вспомогательных переменных начинаются с '`$`'. Любое имя с приставкой '`$`' может использоваться для вспомогательной переменной, если только оно не является предопределенным машинно-зависимым именем регистра, (см. Раздел 8.10 [Регистры], с. 74). (Ссылки на историю значений, напротив, есть *числа*, которым предшествует '`$`'. См. Раздел 8.8 [История значений], с. 72.)

Вы можете сохранить значение во вспомогательной переменной с помощью выражения присваивания, как если бы вы устанавливали переменную в вашей программе. Например:

```
set $foo = *object_ptr
```

сохранит в `$foo` значение объекта, на который указывает `object_ptr`.

Первое использование вспомогательной переменной создает ее, но значением переменной будет `void`, пока вы не присвоите ей новое. С помощью другого присваивания вы можете в любое время изменить значение.

Вспомогательные переменные не имеют фиксированного типа. Вы можете присваивать вспомогательной переменной значение любого типа, включая структуры и массивы, даже если у этой переменной уже было значение другого типа. Будучи использованной в выражении, вспомогательная переменная имеет тип своего текущего значения.

`show convenience`
: Вывести список используемых вспомогательных переменных с их значениями. Сокращается как `show conv`.

Один из способов использования вспомогательных переменных—в качестве увеличивающегося счетчика или продвигающегося указателя. Например, чтобы напечатать поле из последовательных элементов массива структур:

```
set $i = 0
print bar[$i++]->contents
```

Повторяйте эту команду нажатием ⟨RET⟩.

Некоторые вспомогательные переменные создаются GDB автоматически, и им присваиваются значения, которые вероятно могут оказаться полезными.

`$_`
: Переменная `$_` устанавливается автоматически командой `x` в последний исследованный адрес (см. Раздел 8.5 [Исследование памяти], с. 65). Другие команды, которые устанавливают адрес по умолчанию для исследования командой `x`, также присваивают `$_` упомянутый адрес; эти команды включают `info line` и `info breakpoint`. Переменная `$_` имеет тип `void *`, если только она не установлена командой `x`; в этом случае она является указателем на тип переменной `$__`.

`$__`
: Переменная `$__` устанавливается автоматически командой `x` в значение, находящееся по последнему исследованному адресу. Ее тип выбирается соответствующим формату, в котором это значение было выведено.

`$_exitcode`
: Переменной `$_exitcode` автоматически присваивается код завершения, когда отлаживаемая программа завершается.

В системах HP-UX, если вы ссылаетесь на функцию или переменную, чье имя начинается со знака доллара, GDB сначала производит поиск пользовательского или системного имени, перед поиском вспомогательной переменной.

8.10 Регистры

В выражениях, вы можете обращаться к содержимому машинных регистров, обозначая их как переменные с именами, начинающимся с '$'. Имена регистров различаются от машины к машине; для вывода имен регистров, используемых на вашей машине, воспользуйтесь командой `info registers`.

`info registers`
: Вывести имена и содержимое всех регистров, кроме регистров с плавающей точкой (в выбранном кадре стека).

`info all-registers`
: Вывести имена и содержимое всех регистров, включая регистры с плавающей точкой.

`info registers` *имя-рег* `...`
: Выводит *относительное* значение каждого из указанных в *имя-рег* регистров. Как подробно обсуждается ниже, значения регистров обычно относятся к выбранному кадру стека. *Имя-рег* может быть любым допустимым на вашей машине именем регистра, с '$' в начале имени или без.

GDB распознает четыре “стандартных” имени регистров, которые доступны (в выражениях) на большинстве машин—если только они не конфликтуют с каноническим для архитектуры обозначением регистров. Названия регистров `$pc` и `$sp` используются для регистра счетчика программы и указателя вершины стека. `$fp` используется как имя регистра, содержащего указатель на текущий кадр стека, а `$ps`—как имя регистра, содержащего состояние процессора. Например, вы можете вывести счетчик программы в шестнадцатеричной записи с помощью

```
p/x $pc
```

или вывести следующую исполняемую инструкцию, используя

```
x/i $pc
```

или увеличить указатель вершины стека на четыре[2] с помощью

```
set $sp += 4
```

Когда возможно, эти четыре стандартных имени регистров доступны на вашей машине, даже если она имеет другую каноническую мнемонику, если не возникает конфликта. Команда `info registers` показывает канонические имена. В SPARC, например, `info registers` отображает регистр состояния процессора как `$psr`, но вы также можете называть его `$ps`; а на машинах, базирующихся на платформе x86, `$ps` является синонимом для регистра EFLAGS.

Когда регистр изучается таким образом, GDB всегда рассматривает содержимое обычного регистра как целое. Некоторые машины имеют специальные регистры, которые могут содержать только значение с плавающей точкой; их значения трактуются как величины с плавающей точкой. Не существует способа сослаться на содержимое обычного регистра как на величину с плавающей точкой (хотя вы можете распечатать его значение командой *print* как величину с плавающей точкой, используя ‘`print/f $`*имя-рег*’).

Некоторые регистры имеют различные “необработанные” и “виртуальные” форматы данных. Это означает, что формат данных, в котором операционная система сохраняет содержимое регистра, не совпадает с тем, который обычно воспринимается вашей программой. Например, регистры сопроцессора с плавающей точкой 68881 всегда сохраняются в “расширенном” (необработанном) формате, но все программы на Си работают с “двойным” (виртуальным) форматом. В подобных случаях, GDB обычно работает только с виртуальным форматом (форматом, имеющим смысл в вашей программе), но команда `info registers` выводит данные в обоих форматах.

Обычно значения регистров относятся к выбранному кадру стека (см. Раздел 6.3 [Выбор кадра], с. 52). Это значит, что вы получаете значение, которое содержалось бы в регистре, если бы произошел выход из всех внутренних кадров стека и их сохраненные регистры были бы восстановлены. Для того чтобы увидеть истинное содержимое аппаратных регистров, вы должны выбрать самый внутренний кадр (с помощью ‘`frame 0`’).

Однако, GDB, исходя из машинного кода, сгенерированного вашим компилятором, должен установить, где сохранены регистры. Если некоторые регистры не сохранены, или если GDB не в состоянии найти сохраненные регистры, выбранный кадр стека не имеет значения.

8.11 Аппаратные средства поддержки вычислений с плавающей точкой

В зависимости от конфигурации, GDB может выдать вам больше информации о состоянии аппаратных средств поддержки вычислений с плавающей точкой.

[2] На машинах, где стек растет вниз в памяти (в наши дни, на большинстве машин), это способ удалить одно слово из стека. Это подразумевает, что выбран самый внутренний кадр; когда выбраны другие кадры стека, установка `$sp` не разрешена. Используйте `return` для извлечения целого кадра из стека, вне зависимости от архитектуры машины; смотрите Раздел 11.4 [Возврат из функции], с. 103.

`info float`

Отобразить аппаратно-зависимую информацию о модуле поддержки вычислений с плавающей точкой. Ее точное содержание и размещение зависит от микросхемы поддержки вычислений с плавающей точкой. В настоящее время, ‘`info float`’ поддерживается на машинах ARM и x86.

9 Использование GDB с различными языками программирования

Хотя языки программирования обычно имеют общие аспекты, их выражения редко выглядят одинаково. Например, в ANSI Си, разыменование указателя `p` осуществляется операцией `*p`, а в Модуле-2 это выполняется как `p^`. Представление (и отображение) значений также может быть различным. Шестнадцатеричные числа в Си отображаются как '`0x1ae`', в то время как в Модуле-2 они отображаются как '`1AEH`'.

В GDB встроена специальная информация о некоторых языках, которая позволяет описывать действия, подобные упомянутым, на языке вашей программы, и позволяет GDB выводить значения в виде, принятом в языке, на котором написана ваша программа. Язык, которым вы пользуетесь для построения выражений, называется *рабочим языком*.

9.1 Переход от одного языка к другому

Существует два способа управления рабочим языком: либо GDB устанавливает его автоматически, либо вы можете сделать это самостоятельно. Для этих целей вы можете воспользоваться командой `set language`. По умолчанию, при старте GDB устанавливает язык автоматически. Рабочий язык используется чтобы определить, как интерпретируются вводимые вами выражения, как выводятся значения, и так далее.

В дополнение к рабочему языку, каждый исходный файл, с которым работает GDB, имеет свой собственный рабочий язык. Для некоторых форматов объектных файлов компилятор может указывать, на каком языке написан конкретный исходный файл. Однако, чаще всего GDB распознает язык по имени файла. Язык исходного файла определяет, будут ли восстанавливаться имена Си++; таким образом, `backtrace` может показывать каждый кадр в соответствии с исходным языком. Не существует способа установить язык исходного файла из GDB, но вы можете установить язык, ассоциированный с расширением файла. См. Раздел 9.2 [Отображение языка программирования], с. 78.

Наиболее часто эта проблема возникает, когда вы используете программу, такую как `cfront` или `f2c`, которая создает текст на Си, но написана на другом языке. В этом случае нужно сделать, чтобы программа использовала директивы `#line` в выводе текста Си; тогда GDB будет знать правильный язык исходного текста первоначальной программы, и выведет этот исходный код, а не сгенерированный код Си.

9.1.1 Соответствие расширений файлов и языков

Если имя исходного файла заканчивается одним из следующих расширений, то GDB воспринимает это как указание на обозначенный язык.

'`.c`'	Исходный файл Си
'`.C`' '`.cc`' '`.cp`' '`.cpp`' '`.cxx`' '`.c++`'	Исходный файл Си++
'`.f`' '`.F`'	Исходный файл Фортрана
'`.ch`' '`.c186`' '`.c286`'	Исходный файл CHILL
'`.mod`'	Исходный файл Модулы-2

‘.s’
‘.S’ Исходный файл Ассемблера. В действительности, воспринимается почти также, как Си, но GDB не пропускает вводные части функций при пошаговом выполнении.

В дополнение к этому, вы можете установить язык, ассоциированный с расширением имени файла. См. Раздел 9.2 [Отображение языка программирования], с. 78.

9.1.2 Установка рабочего языка

Если вы позволяете GDB устанавливать язык автоматически, выражения в сеансе отладки и в вашей программе интерпретируются одинаково.

По желанию, вы можете установить язык сами. Для этого воспользуйтесь командой ‘`set language` *язык*’, где *язык*—название языка, например, `c` или `modula-2`. Чтобы получить перечень поддерживаемых языков, наберите ‘`set language`’.

Установка языка вручную запрещает GDB автоматически переключать рабочий язык. Это может привести к неприятным последствиям, если вы попытаетесь отладить программу, когда рабочий язык не совпадает с исходным языком, когда выражение допустимо в обоих языках, но означает разные вещи. Например, если текущий исходный файл написан на Си, а в GDB выбрана Модула-2, команда

```
print a = b + c
```

может не привести к ожидаемому результату. В Си это означает сложить `b` и `c` и поместить результат в `a`. Выведенным результатом будет значение `a`. В Модуле-2 это означает сравнение `a` с результатом `b+c`, выдающее значение типа `BOOLEAN`.

9.1.3 Распознавание GDB исходного языка

Для автоматической установки GDB рабочего языка, используйте ‘`set language local`’ или ‘`set language auto`’. Тогда GDB будет определять рабочий язык автоматически. То есть при остановке вашей программы в кадре стека (обычно, в точке останова), GDB устанавливает рабочий язык в тот, который записан для функции в этом кадре. Если язык для кадра неизвестен (то есть, если функция или блок, соответствующие кадру, были определены в исходном файле, не имевшем распознаваемого расширения), текущий рабочий язык не изменяется, а GDB выдает предупреждающее сообщение.

Для большинства программ, которые написаны целиком на одном языке, автоматическая установка языка может показаться ненужной. Однако, в основной программе, написанной на одном исходном языке, могут использоваться программные модули и библиотеки, написанные на другом исходном языке. Использование в этом случае ‘`set language auto`’ освобождает вас от установки рабочего языка вручную.

9.2 Отображение языка программирования

Следующие команды помогают вам определить, какой язык является рабочим, а также на каком языке были написаны исходные файлы.

`show language`
Отобразить текущий рабочий язык. Это тот язык, который вы можете использовать в командах типа `print` для построения и вычисления выражений, в которые могут входить переменные вашей программы.

`info frame`
Отобразить исходный язык для данного кадра стека. Этот язык становится рабочим, если используется идентификатор из этого кадра. См. Раздел 6.4 [Информация о кадре], с. 53, для дополнительной информации.

`info source`
: Отобразить исходный язык данного исходного файла. См. Глава 10 [Исследование таблицы символов], с. 97, для получения дополнительной информации.

При необычных обстоятельствах, у вас могут быть исходные файлы с расширениями, не входящими в стандартный список. Вы можете явно установить расширение, ассоциированное с языком:

`set extension-language` *.расш язык*
: Установить соответствие исходных файлов с расширением *.расш* с исходным языком *язык*.

`info extensions`
: Перечислить все расширения имен файлов и соответствующие им языки.

9.3 Проверка диапазона и принадлежности типу

> *Предупреждение:* В этот выпуск включены команды GDB для проверки диапазона и соответствия типов, но они пока не оказывают никакого действия. Этот раздел описывает их предполагаемые возможности.

Некоторые языки обеспечивают защиту от достаточно общих ошибок с помощью набора проверок времени компиляции и времени выполнения. Это включает проверку типов аргументов функций и операторов и обеспечивает проверку математического переполнения во время выполнения. Проверки такого рода помогают убедиться в корректности программы после ее компиляции путем устранения несоответствия типов, и предоставляя активную проверку ошибок выхода за пределы диапазона во время выполнения.

По вашему желанию, GDB может проводить подобные проверки. Хотя GDB не проверяет операторы вашей программы, он может, например, проверять выражения, введенные непосредственно в GDB для вычисления командой `print`. Как и в случае рабочего языка, GDB может также автоматически решить, выполнять проверку или нет, основываясь на исходном языке вашей программы. См. Раздел 9.4 [Поддерживаемые языки], с. 81, для информации об установках по умолчанию для поддерживаемых языков.

9.3.1 Краткий обзор проверки соответствия типов

Некоторые языки, такие как Модула-2, должны иметь жесткое соответствие типов, то есть аргументы операторов и функций должны иметь правильный тип, в противном случае возникает ошибка. Описанные здесь проверки предотвращают ошибки несоответствия типов, которые могут вызвать ошибки времени выполнения. Например,

```
1 + 2 ⇒ 3
```

но

```
error  1 + 2.3
```

Во втором примере ошибка, потому что `CARDINAL` 1 не совместим по типу с `REAL` 2.3.

Для выражений, используемых вами в командах GDB, вы можете указать GDB не производить проверку; или же рассматривать любое несоответствие как ошибку и прекращать обработку выражения; или только выводить предупреждение в случае возникновения несоответствия, но вычислять выражение в любом случае. В последнем случае, GDB вычисляет выражения, подобные второму примеру, но также выводит предупреждающее сообщение.

Даже если вы отключили проверку типов, GDB может прекратить обработку выражения по другим причинам, связанным с типами. Например, GDB не знает, как сложить `int` и `struct foo`. Такие типы ошибок не имеют ничего общего с используемым языком и обычно возникают из выражений, подобных описанному выше, которые нет смысла вычислять.

Каждый язык определяет степень строгости контроля типов. Например, как Модула-2, так и Си требуют, чтобы аргументы арифметических операций были числами. В Си, перечисляемые типы и указатели могут быть представлены в виде чисел, так что они допустимы в качестве аргументов математических операторов. См. Раздел 9.4 [Поддерживаемые языки], с. 81, для более подробного обсуждения конкретных языков.

GDB предоставляет некоторые дополнительные команды для контроля проверки типов:

`set check type auto`
: Включить или отключить контроль типов, в зависимости от текущего рабочего языка. См. Раздел 9.4 [Поддерживаемые языки], с. 81, для установок по умолчанию для каждого языка.

`set check type on`
`set check type off`
: Включить или отключить контроль типов, пренебрегая установкой по умолчанию для текущего рабочего языка. Вывести предупреждающее сообщение, если установка не соответствует используемой по умолчанию. Если возникает несоответствие типов во время вычисления выражения при включенном контроле типов, GDB выводит сообщение и прерывает вычисление выражения.

`set check type warn`
: При возникновении несоответствия типов вывести предупреждающее сообщение, но попытаться вычислить выражение. Вычисление выражения все же может быть невозможным по другим причинам. Например, GDB не может складывать числа со структурами.

`show type` Показать текущую установку проверки типов, а также была ли она установлена GDB автоматически.

9.3.2 Краткий обзор проверки диапазона

В некоторых языках (например, в Модуле-2), выход за границы диапазона типа считается ошибкой; эти ошибки отслеживаются с помощью контроля времени выполнения. Эти проверки диапазона служат для того, чтобы избежать переполнения при вычислениях и не допустить превышения индексами элементов массива границ индексации.

В выражениях, используемых вами в командах GDB, вы можете указать GDB обрабатывать ошибки диапазона одним из трех способов: игнорировать их, всегда рассматривать как ошибки и прерывать обработку выражения, или выводить предупреждение и продолжать вычисление выражения.

Ошибки диапазона могут возникать при числовом переполнении, при превышении границы индексации массива или при использовании константы, не принадлежащей ни к одному типу. Однако, некоторые языки не считают переполнение ошибкой. Во многих реализациях Си, математическое переполнение вызывает “циклический переход” к меньшему значению. Например, если m—наибольшее целое значение, а s—наименьшее, то

$$m \texttt{ + 1} \Rightarrow s$$

Это также является специфичным для конкретных языков, а в некоторых случаях—для отдельного компилятора или машины. Для дальнейших сведений по отдельным языкам, См. Раздел 9.4 [Поддерживаемые языки], с. 81.

GDB обеспечивает некоторые дополнительные команды для контроля проверки диапазона:

`set check range auto`
: Включить или отключить контроль диапазона, в зависимости от текущего рабочего языка. См. Раздел 9.4 [Поддерживаемые языки], с. 81, для получения информации об установках по умолчанию для каждого языка.

`set check range on`
`set check range off`
: Включить или отключить контроль типов, пренебрегая установкой по умолчанию для текущего рабочего языка. Если установка не соответствует используемой по умолчанию, выводится предупреждающее сообщение. Если происходит ошибка диапазона и контроль включен, выводится сообщение и вычисление выражения прерывается.

`set check range warn`
: При выявлении ошибки диапазона, GDB выведет предупреждающее сообщение, но попытается вычислить выражение. Тем не менее, вычисление выражения может быть невозможным по другим причинам, таким как обращение к памяти, которой процесс не владеет (типичный пример для многих систем Unix).

`show range`
: Показать текущую установку проверки диапазона и была ли она установлена GDB автоматически.

9.4 Поддерживаемые языки

GDB поддерживает Си, Си++, Фортран, Java, Chill, ассемблер и Модула-2. Некоторые возможности GDB могут быть задействованы в выражениях независимо от используемого языка: операторы GDB `@` и `::` и конструкция '`{тип}адрес`' (см. Раздел 8.1 [Выражения], с. 61) могут быть использованы в конструкциях любого поддерживаемого языка.

Следующие разделы подробно описывают, до какой степени каждый из исходных языков поддерживается GDB. Эти разделы не задумывались как учебники или руководства по языкам; они лишь служат справочным руководством по тому, что допускает анализатор выражений GDB, и как должны выглядеть входные и выходные форматы в различных языках. Существует много хороших книг по каждому из этих языков; пожалуйста, загляните в них, если вам нужен учебник или справочник по языку.

9.4.1 Си и Си++

Поскольку Си и Си++ тесно связаны, многие возможности GDB применимы к ним обоим. Когда это имеет место, мы обсуждаем эти языки вместе.

Средства отладки Си++ обеспечиваются совместно компилятором Си++ и GDB. Следовательно, для эффективной отладки программы на Си++, вам следует пользоваться одним из поддерживаемых компиляторов, например GNU `g++`, или компилятором HP ANSI Си++ (`aCC`).

При использовании GNU Си++, для получения наилучших результатов используйте формат отладочной информации stabs. Вы можете выбрать его явно с помощью ключа командной строки '`-gstabs`' или '`-gstabs+`'. Смотрите раздел "Ключи для отладки вашей программы или GNU CC" в *Использование GNU CC*, для дополнительной информации.

9.4.1.1 Операторы Си и Си++

Операторы должны быть определены на значениях определенных типов. Например, `+` определен на числах, но не на структурах. Операторы часто определяются на группах типов.

Для целей Си и Си++, имеют место следующие определения:

- *Целые типы* включают `int` с любыми спецификаторами класса памяти; `char`; `enum`; и, для Си++, `bool`.

- *Типы с плавающей точкой* включают `float`, `double` и `long double` (если поддерживается целевой платформой).
- *Типы указателей* включают все типы, определенные как `(`*тип*` *)`.
- *Скалярные типы* включают все вышеперечисленные типы.

Поддерживаются следующие операторы, перечисленные здесь в порядке возрастания приоритета:

`,` Запятая, или оператор последовательного выполнения. Выражения, разделенные в списке запятыми, вычисляются слева направо; результатом всего выражения является результат, полученный последним.

`=` Присваивание. Значение выражения приваивания—присвоенное значение. Определено на скалярных типах.

опер`=` Используется в выражениях в форме *a опер*`=` *b* и преобразовывается в *a* `=` *a опер b*. *опер*`=` и `=` имеют одинаковый приоритет. *опер* может быть одним из операторов `|`, `^`, `&`, `<<`, `>>`, `+`, `-`, `*`, `/`, `%`.

`?:` Тернарный оператор. *a* `?` *b* `:` *c* истолковывается так: если *a*, то *b*, иначе *c*. *a* должно быть целого типа.

`||` Логическое или. Определено на целых типах.

`&&` Логическое и. Определено на целых типах.

`|` Побитовое или. Определено на целых типах.

`^` Побитовое исключающее или. Определено на целых типах.

`&` Побитовое и. Определено на целых типах.

`==`, `!=` Равенство и неравенство. Определено на скалярных типах. Значение этих выражений отлично от нуля для истины и 0 для лжи.

`<`, `>`, `<=`, `>=`
Меньше чем, больше чем, меньше или равно, больше или равно. Определено на скалярных типах. Значение этих выражений равно 0 для лжи и отлично от нуля для истины.

`<<`, `>>` Левый сдвиг и правый сдвиг. Определено на целых типах.

`@` Оператор GDB создания “искусственного массива” (см. Раздел 8.1 [Выражения], с. 61).

`+`, `-` Сложение и вычитание. Определено на целочисленных типах, типах с плавающей точкой и указателях.

`*`, `/`, `%` Умножение, деление и остаток. Умножение и деление определены на целочисленных типах и типах с плавающей точкой. Остаток определен на целочисленных типах.

`++`, `--` Инкремент и декремент. При появлении перед переменной, операция выполняется прежде, чем переменная используется в выражении; при появлении после переменной, значение переменной используется до выполнения операции.

`*` Разыменование указателя. Определено для указателей. Имеет тот же приоритет, что и `++`.

`&` Оператор получения адреса. Определен на переменных. Имеет тот же приоритет, что и `++`.

Для отладки Си++, GDB реализует использование ‘`&`’ независимо от того, что позволяет сам язык Си++: вы можете использовать ‘`&(&`*ссылка*`)`’ (или просто ‘`&&`*ссылка*’), чтобы исследовать адрес, по которому хранится переменная-ссылка Си++ (объявленная с помощью ‘`&`*ссылка*’).

`-` Унарный минус. Определен на целочисленных типах и типах с плавающей точкой. Имеет тот же приоритет, что и `++`.

`!` Логическое отрицание. Определено на целочисленных типах. Имеет тот же приоритет, что и `++`.

`~` Оператор побитового дополнения. Определен на целочисленных типах. Имеет тот же приоритет, что и `++`.

`., ->` Элемент структуры и указатель на элемент структуры. Для удобства, GDB считает их эквивалентными, определяя, следует ли разыменовывать указатель, основываясь на сохраненной информации о типах. Определен на данных структуры (`struct`) и объединения (`union`).

`.*, ->*` Разыменовывание указателя на элемент.

`[]` Индексация массива. *a*`[`*i*`]` определяется как `*(`*a*`+`*i*`)`. Имеет тот же приоритет, что и `->`.

`()` Список параметров функции. Имеет тот же приоритет, что и `->`.

`::` Оператор Си++ определения области видимости. Определен на типах `struct`, `union` и `class`.

`::` Двойное двоеточие также представляет оператор GDB области видимости (см. Раздел 8.1 [Выражения], с. 61). Имеет тот же приоритет, что и `::`, описанный выше.

Если оператор переопределен в пользовательском коде, GDB обычно пытается выполнить переопределенную версию, а не использовать предопределенное значение оператора.

9.4.1.2 Константы Си и Си++

GDB позволяет вам выражать константы Си и Си++ следующими способами:

- Целочисленные константы—это последовательности цифр. Восьмеричные константы начинаются с '`0`' (с нуля), а шестнадцатеричные константы—с '`0x`' или '`0X`'. Константы также могут заканчиваться буквой '`l`', указывая, что значение константы должно рассматриваться как длинное (`long`).
- Константы с плавающей точкой—это последовательность цифр, за которой следует десятичная точка, другая последовательность цифр, и, возможно, порядок. Порядок указывается в форме '`e`[[`+`]|`-`]*nnn*', где *nnn*—другая последовательность цифр. Для положительных порядков '`+`' является необязательным. Константа с плавающей точкой может также заканчиваться буквой '`f`' или '`F`', это указывает на то, что константа должна рассматриваться как `float` (в отличие от `double` по умолчанию), или буквой '`l`' или '`L`', что указывает на константу типа `long double`.
- Перечисляемые константы состоят из перечисляемых идентификаторов, или их целочисленных эквивалентов.
- Символьные константы—это одиночный символ, заключенный в одиночные кавычки (’), или число—порядковое значение соответствующего символа (обычно его значение ASCII). Внутри кавычек, одиночный символ может быть представлен либо буквой, либо *экранирующей последовательностью*, которая имеет форму '*nnn*', где *nnn* является восьмеричным представлением порядкового значения символа; или форму '*x*', где '*x*'—специальный предопределенный знак, например, '`\n`' для знака новой строки.
- Строковые константы—последовательность символьных констант (без одиночных кавычек), заключенная в двойные кавычки (`"`). Туда могут входить любые допустимые символьные константы (как описано выше). Двойным кавычкам внутри строки должна предшествовать обратная косая черта, так что '`"a\"b’c"`', например, является строкой из пяти символов.

- Константы-указатели представляют собой целочисленные значения. Вы можете также записывать указатели на константы, используя оператор Си '`&`'.
- Константы-массивы—заключенные в фигурные скобки ('`{`' и '`}`') списки элементов, разделенные запятыми; например, '`{1,2,3}`' является массивом с тремя целочисленными элементами, '`{{1,2}, {3,4}, {5,6}}`' является массивом размерности три на два, и '`{&"hi", &"there", &"fred"}`' является трехэлементным массивом указателей.

9.4.1.3 Выражения Си++

Обработчик выражений GDB может интерпретировать большинство выражений Си++.

Предупреждение: GDB может отлаживать программы на Си++ только если вы используете подходящий компилятор. Обычно, отладка Си++ зависит от использования дополнительной отладочной информации в таблице символов, и, таким образом, требует специальной поддержки. В частности, если ваш компилятор генерирует a.out, MIPS ECOFF, RS/6000 XCOFF, или ELF с расширениями stabs к таблице символов, все эти средства доступны. (С GNU CC вы можете использовать ключ '`-gstabs`', чтобы явно запросить расширения отладки stabs). С другой стороны, если формат объектного кода—стандартный COFF или DWARF в ELF, значительная часть поддержки Си++ в GDB *не* работает.

1. Допускаются вызовы функций-членов; вы можете использовать выражения типа

   ```
   count = aml->GetOriginal(x, y)
   ```

2. Пока функция-член активна (в выбранном кадре стека), вашим выражениям доступно то же пространство имен, что и функции-члену; то есть GDB допускает неявные ссылки на указатель экземпляра класса `this` по тем же правилам, что и Си++.
3. Вы можете вызывать перегруженные функции; GDB производит вызов функции с правильным определением, но с некоторыми ограничениями. GDB не совершает преобразования, для выполнения которых требуются преобразования типов, определенные пользователем, вызовы конструкторов или конкретизации не существующих в программе шаблонов. Он также не может обрабатывать списки с неопределенным числом аргументов или аргументы со значениями по умолчанию.

   Он производит преобразования и расширения целочисленных типов, расширения типов с плавающей точкой, арифметические преобразования, преобразования указателей, преобразования класса объекта в базовые классы и стандартные преобразования, например функции или массива к указателю; это требует точного совпадения числа аргументов функции.

   Разрешение перегруженных имен производится всегда, если не указано `set overload-resolution off`. См. Раздел 9.4.1.7 [Возможности GDB для Си++], с. 85.

   Вы должны указать `set overload-resolution off`, чтобы задать функцию явно при вызове перегруженной функции, как в примере

   ```
   p 'foo(char,int)'('x', 13)
   ```

   Возможности GDB для завершения команд могут упростить это; смотрите Раздел 3.2 [Завершение команд], с. 15.
4. GDB понимает переменные, объявленные как ссылки Си++; вы можете использовать их в выражениях, точно как вы делаете в исходном тексте Си++—они автоматически разыменовываются.

   В списке параметров, показываемом GDB при отображении кадра стека, значения переменных-ссылок не отображаются (в отличие от других переменных); это позволяет избежать неудобств из-за того, что ссылки часто используются для больших структур. *Адрес* переменной-ссылки всегда выводится, если только вы не установили '`set print address off`'.

5. GDB поддерживает оператор Си++ определения области видимости имени `::`—ваши выражения могут использовать его так же, как в вашей программе. Так как одна область видимости может быть определена внутри другой, вы можете при необходимости неоднократно использовать `::`, например, в выражении типа '*обл1*`::`*обл2*`::`*имя*'. GDB также позволяет определить область видимости имени путем ссылки на исходный файл, при отладке как Си, так и Си++ (см. Раздел 8.2 [Переменные программы], с. 62).

Кроме того, при использовании с компилятором HP Си++, GDB правильно поддерживает вызов виртуальных функций, вывод виртуальный баз объектов, вызов функций в базовом подобъекте, приведение объектов и выполнение операторов, определенных пользвателем.

9.4.1.4 Значения Си и Си++ по умолчанию

Если вы разрешаете GDB устанавливать проверки диапазона и принадлежности типу автоматически, обе они по умолчанию *отключены*, если рабочий язык изменяется на Си или Си++. Это происходит независимо от того, выбираете рабочий язык вы или GDB.

Если вы разрешаете GDB устанавливать язык автоматически, он распознает исходные файлы, чьи имена заканчиваются расширением '`.c`', '`.C`' или '`.cc`', и так далее, и когда GDB начинает обработку кода, скомпилированного из одного из этих файлов, он устанавливает рабочий язык в Си или Си++. См. Раздел 9.1.3 [Распознавание GDB рабочего языка], с. 78, для более подробного обсуждения.

9.4.1.5 Проверки диапазона и принадлежности типу в Си и Си++

Когда GDB производит разбор выражений Си или Си++, по умолчанию проверки соответствия типов не проводятся. Однако, если вы их включите, GDB считает типы двух переменных эквивалентными, если:

- Обе переменные структурированы и имеют один и тот же тег структуры, объединения или перечисления.
- Имена типов обеих переменных совпадают или были объявлены эквивалентными через `typedef`.

Проверка диапазона, если она включена, выполняется для математических операций. Индексы массивов не проверяются, так как они часто применяются для индексирования указателей, которые сами по себе массивами не являются.

9.4.1.6 GDB и Си

Команды `set print union` и `show print union` применимы к типу `union`. При установке в '`on`', любые объединения, находящиеся внутри структуры или класса, также выводятся. В противном случае, они отображаются как '`{...}`'.

Оператор `@` помогает при отладке динамических массивов, сформированных с помощью указателей и функции выделения памяти. См. Раздел 8.1 [Выражения], с. 61.

9.4.1.7 Возможности GDB для Си++

Некоторые команды GDB особенно полезны при использовании с Си++, а некоторые разработаны специально для него. Ниже приведено их краткое описание:

меню точки останова
: Когда вы хотите установить точку останова в перегруженной функции, меню точки останова GDB помогает вам указать, какое определение функции вам нужно. См. Раздел 5.1.8 [Меню точки останова], с. 42.

`rbreak` *рег-выр*
: Установка точек останова при помощи регулярных выражений полезна при использовании перегруженных функций, не являющихся членами специальных классов. См. Раздел 5.1.1 [Установка точек останова], с. 32.

`catch throw`
`catch catch`
: Отлаживайте обработку исключений Си++ с помощью этих команд. См. Раздел 5.1.3 [Установка точек перехвата], с. 37.

`ptype` *имя-типа*
: Вывести отношения наследования вместе с другой информацией для типа *имя-типа*. См. Глава 10 [Исследование таблицы символов], с. 97.

`set print demangle`
`show print demangle`
`set print asm-demangle`
`show print asm-demangle`
: Управляет отображением символов Си++ в их исходной форме, как при выводе кода в виде исходного текста Си++, так и при выводе результата дисассемблирования. См. Раздел 8.7 [Параметры вывода], с. 68.

`set print object`
`show print object`
: Выбрать, выводить производные (реальные) или описанные типы объектов. См. Раздел 8.7 [Параметры вывода], с. 68.

`set print vtbl`
`show print vtbl`
: Управляет форматом вывода таблиц виртуальных функций. См. Раздел 8.7 [Параметры вывода], с. 68. (Команды `vtbl` не работают для программ, скомпилированных компилятором HP ANSI Си++ (`aCC`).)

`set overload-resolution on`
: Включить разрешение перегруженных символов при вычислении выражений Си++. Значение по умолчанию `on`. Для перегруженных функций, GDB вычисляет аргументы и ищет функции, чьи сигнатуры удовлетворяют типам аргументов, используя стандартные правила преобразования Си++ (смотрите Раздел 9.4.1.3 [Выражения Си++], с. 84, для дополнительной информации). Если GDB не может найти такие функции, он выводит сообщение.

`set overload-resolution off`
: Отключить разрешение перегруженных символов при вычислении выражений Си++. Для перегруженных функций, не являющихся функциями-членами класса, GDB выбирает функцию с указанным именем, которую он первой находит в таблице символов, в не зависимости от того, правильного типа ее аргументы или нет. Для перегруженных функций, являющихся функциями-членами класса, GDB ищет функцию, чья сигнатура *точно* совпадает с типами аргументов.

Перегруженные имена смиволов
: Вы можете указать конкретное определение перегруженного символа, используя ту же запись, что и для объявления таких символов в Си++: введите *символ*`(`*типы*`)` вместо просто *символ*. Вы также можете воспользоваться средствами завершения слова командной строки GDB, чтобы вывести список возможных вариантов, или чтобы завершить набор за вас. См. Раздел 3.2 [Завершение команд], с. 15, для подробного обсуждения, как это сделать.

9.4.2 Модула-2

Расширения, сделанные в GDB для поддержки Модулы-2, поддерживаются только для программ, скомпилированных GNU компилятором Модулы-2 (который сейчас разрабатывается). Другие компиляторы Модулы-2 в настоящее время не поддерживаются, и попытка отладки исполняемых программ, полученных ими, скорее всего приведет к ошибке при считывании GDB таблицы символов этой программы.

9.4.2.1 Операторы Модулы-2

Операторы должны быть определены на значениях определенных типов. Например, `+` определен на числах, но не на структурах. Операторы часто определяются на группах типов. Для целей Модулы-2, имеют место следующие определения:

- *Целые типы* состоят из `INTEGER`, `CARDINAL` и их поддиапазонов.
- *Символьные типы* состоят из `CHAR` и его поддиапазонов.
- *Типы с плавающей точкой* состоят из `REAL`.
- *Типы-указатели* состоят из всего, объявленного как `POINTER TO` *тип*.
- *Скалярные типы* включают все вышеперечисленное.
- *Типы-множества* состоят из типов `SET` и `BITSET`.
- *Булевый тип* состоит из `BOOLEAN`.

Поддерживаются следующие операторы; они представлены в порядке возрастания приоритета:

`,` Разделитель аргументов функции или индексов массива.

`:=` Присваивание. Значением *перем* `:=` *знач* является *знач*.

`<`, `>` Меньше чем, больше чем для целочисленных типов, типов с плавающей точкой и перечислимых типов.

`<=`, `>=` Меньше или равно, больше или равно. Определено на целочисленных типах, типах с плавающей точкой и перечислимых типах. Включение для множеств. Такой же приоритет, как у `<`.

`=`, `<>`, `#` Равенство и два способа выражения неравенства; допустимо на скалярных типах. Такой же приоритет, как у `<`. В сценариях GDB, для неравенства допустимо только `<>`, так как `#` конфликтует со знаком комментария.

`IN` Установка принадлежности. Определено на множествах и типах их элементов. Такой же приоритет, как у `<`.

`OR` Дизъюнкция (логическое ИЛИ). Определена на булевых типах.

`AND`, `&` Конъюнкция (логическое И). Определена на булевых типах.

`@` Оператор “искусственного массива” GDB (см. Раздел 8.1 [Выражения], с. 61).

`+`, `-` Сложение и вычитание на целочисленных типах и типах с плавающей точкой, или объединение и разность на множественных типах.

`*` Умножение на целочисленных типах и типах с плавающей точкой, или пересечение на типах-множествах.

`/` Деление на типах с плавающей точкой. Симметрическая разность множеств на типах-множествах. Такой же приоритет, как у `*`.

`DIV`, `MOD` Целочисленное деление и остаток. Определены на целочисленных типах. Такой же приоритет, как у `*`.

`-` Отрицание. Определено на данных типов `INTEGER` и `REAL`.

`^` Разыменовывание указателя. Определено на типах-указателях.

`NOT` Булево отрицание. Определено на булевых типах. Такой же приоритет, как у `^`.

`.` Селектор полей `RECORD`. Определен для данных типа `RECORD`. Такое же приоритет, как у `^`.

`[]` Индексация массива. Определена для данных типа `ARRAY`. Такой же приоритет, как у `^`.

`()` Список параметров процедуры. Определен на объектах `PROCEDURE`. Такой же приоритет, как у `^`.

`::`, `.` Операторы GDB и Модулы-2 определения области видимости.

Предупреждение: Множества и операции над ними еще не поддерживаются, так что GDB трактует использование оператора `IN` или операторов `+`, `-`, `*`, `/`, `=`, , `<>`, `#`, `<=`, и `>=` на множествах как ошибку.

9.4.2.2 Встроенные функции и процедуры

Модула-2 также делает доступными несколько встроенных процедур и функций. При их описании, используются следующие метапеременные:

a представляет переменную типа `ARRAY`.

c представляет константу или переменную типа `CHAR`.

i представляет переменную или константу целого типа.

m представляет идентификатор, принадлежащий множеству. Обычно используется в одной функции с метапеременной *s*. Тип *s* должен быть `SET OF` *метатип* (где *метатип*—тип *m*).

n представляет переменную или константу целого типа или типа с плавающей точкой.

r представляет переменную или константу типа с плавающей точкой.

t представляет тип.

v представляет переменную.

x представляет переменную или константу одного из нескольких типов. Смотрите пояснение к функции для дополнительной информации.

Ниже описаны все встроенные процедуры Модулы-2, возвращающие результат.

`ABS(`*n*`)` Возвращает абсолютное значение *n*.

`CAP(`*c*`)` Если *c*—символ нижнего регистра, процедура возвращает его эквивалент в верхнем регистре, иначе возвращает сам аргумент.

`CHR(`*i*`)` Возвращает символ, порядковое значение которого есть *i*.

`DEC(`*v*`)` Уменьшает значение переменной *v* на единицу. Возвращает новое значение.

`DEC(`*v*`,`*i*`)` Уменьшает значение переменной *v* на *i*. Возвращает новое значение.

`EXCL(`*m*`,`*s*`)`
Удаляет элемент *m* из множества *s*. Возвращает новое множество.

`FLOAT(`*i*`)` Возвращает эквивалент целого числа *i* в формате с плавающей точкой.

`HIGH(`*a*`)` Возвращает индекс последнего элемента *a*.

`INC(`*v*`)` Увеличивает значение переменной *v* на единицу. Возвращает новое значение.

`INC(`*v*`,`*i*`)` Увеличивает значение переменной *v* на *i*. Возвращает новое значение.

`INCL(`*m*`,`*s*`)`
Добавляет элемент *m* в множество *s*, если его там еще нет. Возвращает новое множество.

`MAX(`*t*`)` Возвращает максимальное значение типа *t*.

`MIN(`*t*`)` Возвращает минимальное значение типа *t*.

`ODD(`*i*`)` Возвращает булево значение TRUE, если число *i* нечетно.

`ORD(`*x*`)` Возвращает порядковое значение своего аргумента. Например, порядковое значение символа—его ASCII-значение (на машинах, поддерживающих набор символов ASCII). *x* должна принадлежать упорядоченному типу, что включает целочисленные, символьный и перечислимый типы.

`SIZE(`*x*`)` Возвращает размер аргумента. *x* может быть переменной или типом.

`TRUNC(`*r*`)` Возвращает целую часть *r*.

`VAL(`*t*`,`*i*`)` Возвращает элемент типа *t*, порядковое значение которого есть *i*.

> *Предупреждение:* Множества и операции над ними еще не поддерживаются, так что GDB рассматривает использование процедур `INCL` и `EXCL` как ошибку.

9.4.2.3 Константы

GDB позволяет вам выражать константы Модулы-2 следующими способами:

- Целые константы являются просто последовательностью цифр. При использовании в выражении, константа интерпретируется так, чтобы быть совместимой по типу с остальной частью выражения. Шестнадцатеричные целые числа определяются окончанием '`H`', а восьмеричные— окончанием '`B`'.
- Константы с плавающей точкой задаются как последовательность цифр, за которой следует десятичная точка и другая последовательность цифр. Необязательный порядок может быть задан в форме '`E`[`+`|`-`]*nnn*', где '[`+`|`-`]*nnn*' и есть желаемый порядок. Все цифры константы с плавающей точкой должны быть десятичными (по основанию 10).
- Символьные константы состоят из одиночных символов, заключенных в пару одинаковых кавычек: либо одиночных (`'`), либо двойных (`"`). Они также могут быть заданы своим порядковым значением (обычно ASCII-значением), за которым следует '`C`'.
- Строковые константы состоят из последовательности символов, окруженных парой одинаковых кавычек: либо одиночных (`'`), либо двойных (`"`). Также допускаются экранирующие последовательности в стиле Си. См. Раздел 9.4.1.2 [Константы Си и Си++], с. 83, для краткого объяснения экранирующих последовательностей.
- Перечислимые константы состоят из перечислимого идентификатора.
- Булевы константы состоят из идентификаторов `TRUE` и `FALSE`.
- Константы-указатели состоят только из целочисленных значений.
- Константы-множества пока не поддерживаются.

9.4.2.4 Установки по умолчанию Модулы-2

Если проверка диапазона или принадлежности типу устанавливается GDB автоматически, то по умолчанию обе они устанавливаются в `on`, если рабочим языком становится Модула-2. Это происходит независимо от того, кто выбрал рабочий язык—вы или GDB.

Если вы разрешаете GDB выбирать язык автоматически, то при анализе кода, скомпилированного из файла, чье имя оканчивается на '`.mod`', GDB установит рабочим языком Модулу-2. См. Раздел 9.1.3 [Распознавание GDB исходного языка], с. 78, для дополнительной информации.

9.4.2.5 Отклонения от стандарта Модулы-2

Для упрощения отладки программ на Модуле-2 было сделано несколько изменений. В основном, это сделано путем ослабления строгости контроля типов:

- В отличие от стандарта Модулы-2, константы-указатели могут быть сформированы целыми числами. Это позволяет вам изменять переменные-указатели в процессе отладки. (В стандарте Модулы-2, реальный адрес, содержащийся в переменной-указателе, скрыт от вас; его можно изменить лишь прямым присваиванием значения другой переменной-указателя или выражения, возвращающего указатель.)
- Экранирующие последовательности Си могут использоваться в строках и символах, чтобы представить непечатаемые символы. GDB выводит строки со встроенными экранирующими последовательностями. Одиночные непечатаемые символы выводятся с помощью формата '`CHR(`*nnn*`)`'.
- Оператор присваивания (`:=`) возвращает значение своего правого аргумента.
- Все встроенные процедуры как изменяют, так *и* возвращают свой аргумент.

9.4.2.6 Проверки диапазона и принадлежности типу Модулы-2

> *Предупреждение:* в этом выпуске, GDB еще не выполняет проверки диапазона и принадлежности типу.

GDB считает две переменные Модулы-2 эквивалентными по типу, если:

- Их типы были объявлены эквивалентными посредством оператора `TYPE` *t1* `=` *t2*
- Они были объявлены на одной и той же строке. (Примечание: Это верно для компилятора GNU Модула-2, но это может не выполнятся для других компиляторов.)

Пока проверка соответствия типов включена, любая попытка скомбинировать переменные не эквивалентных типов является ошибкой.

Проверка диапазона выполняется во всех математических операциях, присваиваниях, при индексации массивов и во всех встроенных функциях и процедурах.

9.4.2.7 Операторы определения области видимости `::` и `.`

Существует несколько тонких различий между операторами области видимости Модулы-2 (`.`) и GDB (`::`). Оба имеют похожий синтаксис:

модуль `.` *идент*
область `::` *идент*

где *область*—имя модуля или процедуры, *модуль*—имя модуля, а *идент*—любой идентификатор, описанный в пределах вашей программы, за исключением другого модуля.

Использование оператора `::` заставляет GDB искать идентификатор *идент* в *области*. Если он в ней не найден, GDB ищет его во всех областях, содержащих *область*.

Использование оператора `.` заставляет GDB искать идентификатор *идент*, который был импортирован из модуля определения *модуль*, в текущей области видимости. В этом операторе считается ошибкой, если идентификатор *идент* не был импортирован из модуля определения *модуль*, или если *идент* не является в нем идентификатором.

9.4.2.8 GDB и Модула-2

Некоторые команды GDB имеют мало смысла при отладке программ на Модуле-2. Пять подкоманд из команд `set print` и `show print` применимы исключительно к Си и Си++: '`vtbl`', '`demangle`', '`asm-demangle`', '`object`' и '`union`'. Первые четыре применимы к Си++, а последняя к типу Си `union`, который не имеет прямого аналога в Модуле-2.

Оператор `@` (см. Раздел 8.1 [Выражения], с. 61) хоть и доступен при использовании любого языка, бесполезен при работе с Модулой-2. Его цель состоит в том, чтобы помочь при отладке *динамических массивов*, которые не могут быть созданы в Модуле-2 в отличие от Си или Си++. Однако, конструкция '`{`*тип*`}`*адр-выр*' все же полезна, так как адрес может быть определен целочисленной константой.

В сценариях GDB, оператор неравенства Модулы-2 `#` интерпретируется как начало комментария. Используйте вместо него `<>`.

9.4.3 Chill

Расширения, сделанные в GDB для поддержки Chill, работают только с программами, созданными компилятором GNU Chill. Другие компиляторы Chill в настоящее время не поддерживаются, и попытка отладить программы, полученные с их помощью, скорее всего приведет к ошибке в тот момент, когда GDB будет считывать таблицу символов выполняемого файла.

Этот раздел охватывает темы, связанные с Chill, и возможности GDB для их поддержки.

9.4.3.1 Как отображаются режимы

Поддержка GDB типов данных (режимов) Chill непосредственно связана с возможностями компилятора GNU Chill, и, следовательно, слегка отличается от стандартной спецификации языка. Вот предоставляемые режимы:

Дискретные режимы:

- *Целочисленные режимы*, которые предопределены как `BYTE, UBYTE, INT, UINT, LONG, ULONG`,
- *Булевский режим*, который предопределен как `BOOL`,
- *Символьный режим*, который предопределен как `CHAR`,
- *Режим-множество*, который отображается ключевым словом `SET`.

  ```
  (gdb) ptype x
  type = SET (karli = 10, susi = 20, fritzi = 100)
  ```

  Если тип является ненумерованным множеством, значения элементов множества опускаются.
- *Режим-диапазон*, который отображается как

  ```
  тип = <базовый-режим>(<нижняя граница> : <верхняя граница>)
  ```

  где `<нижняя граница>, <верхняя граница>` может быть любым дискретным буквенным выражением (например, имена элементов множества).

Режим powerset:

Режим Powerset отображается ключевым словом `POWERSET`, за которым следует режим элемента.

```
(gdb) ptype x
type = POWERSET SET (egon, hugo, otto)
```

Режимы-ссылки:

- *Режим привязанной ссылки*, который отображается ключевым словом `REF`, за которым следует название режима, к которому ссылка привязана.
- *Режим свободной ссылки*, который отображается ключевым словом `PTR`.

Процедурный режим

Процедурный режим отображается в виде `тип = PROC(<список параметров>) <возвращаемый режим> EXCEPTIONS (<список исключений>)`. `<список параметров>` представляет собой список режимов параметров. `<возвращаемый режим>` указывает режим результата процедуры, если она возвращает результат. `<список исключений>` перечисляет все возможные исключения, которые могут быть возбуждены процедурой.

Синхронизационные режимы:

- *Режим события*, который отображается как

  `EVENT (<длина события>)`

  где `<длина события>` является необязательной.
- *Буферный режим*, который отображается как

  `BUFFER (<длина буфера>)<режим елементов буфера>`

  где `(<длина буфера>)` является необязательной.

Режимы времени:

- *Режим длительности*, который предопределен как `DURATION`
- *Режим абсолютного времени*, который предопределен как `TIME`

Вещественные режимы:

Вещественные режимы предопределены как `REAL` и `LONG_REAL`.

Строковые режимы:

- *Режим строки символов*, который отображается как

  `CHARS(<длина строки>)`

  за которым следует ключевое слово `VARYING`, если строковый режим является изменяющимся режимом
- *Режим строки битов*, который отображается как

  `BOOLS(<длина строки>)`

Режим массива:

Режим массива отображается ключевым словом `ARRAY(<диапазон>)`, за которым следует режим элементов (который, в свою очередь, может быть режимом массива).

```
(gdb) ptype x
type = ARRAY (1:42)
          ARRAY (1:20)
             SET (karli = 10, susi = 20, fritzi = 100)
```

Структурный режим

Структурный режим отображается ключевым словом `STRUCT(<список полей>)`. `<список полей>` состоит из имен и режимов полей структуры. Структуры с вариантами имеют ключевое слово `CASE <поле> OF <варианты поля> ESAC` в их списке полей. Так как текущая версия компилятора GNU Chill не реализует обработку тегов (нет проверок времени выполнения вариантных полей, и, следовательно, нет отладочной информации), вывод всегда содержит все вариантные поля.

```
(gdb) ptype str
type = STRUCT (
    as x,
    bs x,
    CASE bs OF
    (karli):
        cs a
    (ott):
        ds x
    ESAC
)
```

9.4.3.2 Местоположения и доступ к ним

Местоположением в Chill является объект, который может содержать значения.

Доступ к значению местоположения обычно производится посредством (описанного) имени местоположения. Вывод удовлетворяет спецификации значений в программах на Chill. То, как значения задаются, является темой следующего раздела, смотрите Раздел 9.4.3.3 [Значения и операции с ними], с. 93.

Псевдо-местоположение `RESULT` (или `result`) может использоваться для отображения или изменения результата процедуры, активной в настоящий момент:

```
set result := EXPR
```

Это делает то же самое, что и действие Chill `RESULT EXPR` (которое в GDB недоступно).

Значения местоположений режима ссылок выводятся, в случае режима свободной ссылки, посредством `PTR(<шестнадцатеричное значение>)`, и с помощью `(REF <режим ссылки>) (<шестнадцатеричное значение>)` в случае привязанной ссылки. `<шестнадцатеричное значение>` представляет адрес, на который указывает ссылка. Для доступа к значению местоположения, указываемого ссылкой, используйте оператор разыменовывания '`->`'.

Значения местоположений процедурного режима отображаются как

```
{ PROC
(<режимы аргументов> ) <возвращаемый режим> } <адрес> <имя
местоположения процедуры>
```

`<режимы аргументов>`—это список режимов, в соответствии со спецификацией параметров процедуры, а `<адрес>` указывает адрес точки входа.

Подструктуры значений строковых режимов, режимов массивов или структур (например, срезы массивов, поля структурных местоположений) доступны при использовании определенных операторов, которые описаны в следующем разделе, смотрите Раздел 9.4.3.3 [Значения и операции с ними], с. 93.

Значение местоположения может быть интерпретировано как имеющее другой режим посредством преобразования местоположений. Это преобразование режимов записывается как `<имя режима>(<местоположение>)`. Пользователь должен учесть, что размеры режимов должны быть равными, в противном случае возникает ошибка. Более того, не производится никаких проверок диапазона местоположения по сравнению с режимом назначения, и, следовательно, результат может быть достаточно обескураживающим.

```
(gdb) print int (s(3 up 4)) XXX TO be filled in !! XXX
```

9.4.3.3 Значения и операции с ними

Значения используются для изменения местоположений, для более подробного изучения сложных структур и для отфильтровывания значимой информации из большого объема

данных. Определено несколько операций (зависящих от режима), которые позволяют проводить подобные изучения. Эти операции применимы не только к значениям-константам, но также и к местоположениям, что может оказаться достаточно полезным при отладке сложных структур. Во время разбора командной строки (например, вычисляя выражение), GDB рассматривает имена местоположений как значения этих местоположений.

Этот раздел описывает, как должны задаваться значения и какие операции допустимо использовать с этими значениями.

`Буквенные значения`
: Буквенные значения определяются также, как в программах GNU Chill. Для подробной спецификации, смотрите главу 1.5 Руководства по реализации GNU Chill.

`Значения-наборы`
: Набор может быть задан как **<имя режима>[<набор>]**, где **<имя режима>** может быть опущено, если режим набора определяется однозначно. Эта однозначность определяется из контекста вычисляемого выражения. **<набор>** может быть одним из:

  - *Набор powerset*
  - *Набор массивов*
  - *Набор структур*

  Наборы-powerset, наборы массивов и наборы структур определяются также, как в программах на Chill.

`Значение элемента строки`
: Значение элемента строки задается как

  <строковое значение>(<индекс>)

  где **<индекс>** является целочисленным выражением. Это дает символьное значение, которое экивалентно символу, указываемому в строке индексом **<индекс>**.

`Значение среза строки`
: Значение среза строки задается как **<значение строки>(<спецификация среза>)**, где **<спецификация среза>** может быть либо диапазоном целых выражений, либо задаваться в виде **<начальное выражение> up <размер>**. **<размер>** обозначает число элементов, которое содержит срез. Полученная величина является строкой, которая является частью указанной строки.

`Значения элементов массива`
: Значение элемента массива указывается как **<величина массива>(<выр>)** и дает величину элемента массива с режимом как у указанного массива.

`Значение среза массива`
: Срез массива задается как **<значение массива>(<спецификация среза>)**, где **<спецификация среза>** может быть диапазоном, определенным либо выражениями, либо как **<начальное выр> up <размер>**. **<размер>** обозначает число элементов массива, которое содержит срез. Получаемое значение есть массив, который является частью указанного.

`Значение поля структуры`
: Значение поля структуры получается как **<значение структуры>.<имя поля>**, где **<имя поля>** указывает имя поля, заданное в определении режима структуры. Режим полученного значения соответствует этому определению режима в определении структуры.

`Значения вызова процедуры`
: Значение вызова процедуры получается из значения, возвращенного процедурой[1].

  Значения местоположений режима длительности представляются буквами `ULONG`.

  Значения местоположений режима-времени выводятся как

  ```
  TIME(<сек>:<нсек>).
  ```

`Значение безаргументного оператора`
: Значение безаргументного оператора получается из значения экземпляра для текущего активного процесса.

`Значения выражений`
: Значение, доставляемое выражением, является результатом вычисления указанного выражения. В случае ошибки (несовместимость режимов, и так далее), вычисление выражения прерывается с соответствующим сообщением об ошибке. Выражение может быть заключено в скобки, что приводит к вычислению этого выражения до любого другого, использующего результат выражения в скобках. GDB поддерживаются следующие операторы:

  `OR, ORIF, XOR`
  `AND, ANDIF`
  `NOT` Логические операторы, определенные на операндах булевого режима.

  `=, /=` Операторы равенства и неравенства, определенные на всех режимах.

  `>, >=`
  `<, <=` Операторы отношения, заданные на предопределенных режимах.

  `+, -`
  `*, /, MOD, REM`
  Арифметические операторы, заданные на предопределенных режимах.

  `-` Оператор изменения знака.

  `//` Оператор соединения строк.

  `()` Оператор повторения строки.

  `->` Оператор ссылки местоположения, который может быть использован либо для получения адреса местоположения (`->loc`), или для разыменовывания ссылки местоположения (`loc->`).

  `OR, XOR`
  `AND`
  `NOT` Операторы режимов powerset и строки битов.

  `>, >=`
  `<, <=` Операторы включения режима powerset.

  `IN` Оператор принадлежности.

[1] Если, например, вызов процедуры используется в выражении, то эта процедура вызывается со всеми своими побочными эффектами. При неаккуратном использовании это может привести к путанице.

9.4.3.4 Проверка диапазона и типов в Chill

GDB считает два режима переменных Chill эквивалентными, если их размеры равны. Это правило применяется рекурсивно для более сложных типов данных. Это означает, что сложные режимы считаются эквивалентными, если режимы всех элементов (которые тоже могут быть сложными, например, массивами, структурами, и так далее) имеют одинаковый размер.

Проверка диапазона производится для всех математических операций, присваиваний, границ индексов массива и всех встроенных процедур.

Строгие проверки типов включаются с помощью команды GDB `set check strong`. Это навязывает строгую проверку диапазона и принадлежности типу для всех действий, где используются конструкции Chill (выражения, встроенные функции, и так далее), в соответствии с семантикой, определенной в спецификации языка z.200.

Все проверки могут быть отключены командой GDB `set check off`.

9.4.3.5 Установки по умолчанию Chill

Если проверки типа и диапазона установлены GDB автоматически, обе они по умолчанию включены, когда рабочий язык переключается на Chill. Это происходит независимо от того, вы выбрали рабочий язык или GDB.

Если вы разрешите GDB устанавливать рабочий язык автоматически, то при попадании в код, скомпилированный из файла, чье имя заканчивается на '`.ch`', он переключает рабочий язык на Chill. См. Раздел 9.1.3 [Распознавание GDB исходного языка], с. 78, для дополнительной информации.

10 Исследование таблицы символов

Команды, описанные в этой главе, позволяют вам получить информацию о символах (именах переменных, функций и типов), определенных в вашей программе. Эта информация присуща тексту вашей программы и не изменяется при ее выполнении. GDB находит эту информацию в таблице символов вашей программы, в файле, определенном при его вызове (см. Раздел 2.1.1 [Выбор файлов], с. 10), или посредством одной из команд управления файлами (см. Раздел 12.1 [Команды для задания файлов], с. 105).

Иногда вам может потребоваться сослаться на символы, содержащие необычные знаки, которые GDB обычно трактует как разделители слов. Наиболее часто это встречается при ссылках на статические переменные в других исходных файлах (см. Раздел 8.2 [Переменные программы], с. 62). Имена файлов записаны в объектных файлах как отладочные символы, но GDB обычно производит разбор типичного имени файла, например ‘`foo.c`’, как три слова: ‘`foo` `.` `c`. Чтобы GDB идентифицировал ‘`foo.c`’ как одно слово, заключите его в одинарные кавычки; например,

```
p 'foo.c'::x
```

ищет значение `x` в области видимости файла ‘`foo.c`’.

`info address` *символ*
: Описывает, где хранятся данные для *символа*. Для регистровой переменной сообщается, в каком регистре она содержится. Для нерегистровой локальной переменной печатается смещение в кадре стека, по которому переменная всегда хранится.

  Заметьте отличие от команды ‘`print &`*символ*’, которая вообще не работает для регистровых переменных, а для локальной переменной из стека печатает точный адрес текущего экземпляра переменной.

`whatis` *выраж*
: Напечатать тип данных выражения *выраж*. На самом деле *выраж* не вычисляется, а присутствующие в нем побочные операции (такие как присваивания или вызовы функций) не выполняются. См. Раздел 8.1 [Выражения], с. 61.

`whatis`
: Вывести тип данных `$`, последней записи в истории значений.

`ptype` *имя-типа*
: Вывести описание типа данных *имя-типа*. *Имя-типа* может быть именем типа или, для кода Си, может иметь форму ‘`class` *имя-класса*’, ‘`struct` *тег-структуры*’, ‘`union` *тег-объединения*’ или ‘`enum` *тег-перечисления*’.

`ptype` *выраж*
`ptype`
: Вывести описание типа выражения *выраж*. `ptype` отличается от `whatis` тем, что выводится детальное описание, а не только имя типа.

  Например, для такого описания переменной:

  ```
  struct complex {double real; double imag;} v;
  ```

  эти две команды выведут следующее:

  ```
  (gdb) whatis v
  type = struct complex
  (gdb) ptype v
  type = struct complex {
      double real;
      double imag;
  }
  ```

  Как и `whatis`, использование `ptype` без параметра относится к типу `$`, последней записи в истории значений.

`info types` *рег-выр*
`info types`
: Вывести краткое описание всех типов, имена которых соответствуют регулярному выражению *рег-выр* (или всех типов вашей программы, если вы используете эту команду без параметра). Каждое полное имя типа сопоставляется так, как если бы оно было полной строкой; таким образом, '`i type value`' выдает информацию обо всех типах в вашей программе, чьи имена включают строку `value`, а '`i type ^value$`' выдает информацию только о типах с полным именем `value`.

  Эта команда отличается от `ptype` следующим: во-первых, как и `whatis`, она не выводит детального описания; во-вторых, она перечисляет все исходные файлы, где определен тип.

`info source`
: Показать имя текущего исходного файла—то есть исходного файла для функции, содержащей текущую точку выполнения, и язык, на котором она написана.

`info sources`
: Вывести имена всех исходных файлов вашей программы, для которых имеется отладочная информация, организуя их в два списка: файлы с уже прочитанными символами, и файлы, символы которых будут прочитаны, когда потребуется.

`info functions`
: Вывести имена и типы данных всех определенных функций.

`info functions` *рег-выр*
: Вывести имена и типы данных всех определенных функций, чьи имена удовлетворяют регулярному выражению *рег-выр*. Так, '`info fun step`' находит все функции, имена которых содержат `step`; '`info fun ^step`' находит функции с именами, начинающимися со `step`.

`info variables`
: Напечатать имена и типы данных всех переменных, объявленных вне функций (то есть исключая локальные переменные).

`info variables` *рег-выр*
: Вывести имена и типы данных всех переменных (кроме локальных), имена которых удовлетворяют регулярному выражению *рег-выр*.

  Некоторые системы допускают замещение отдельных объектных файлов, составляющих вашу программу, без ее остановки и перезапуска. Например, в VxWorks вы можете просто перекомпилировать дефектный объектный файл и продолжить выполнение. Если вы работаете в одной из таких систем, вы можете позволить GDB перезагрузить символы для автоматически пересобранных модулей:

  `set symbol-reloading on`
  : Заменить опеделения символов для соответствующего исходного файла, когда объектный файл с определенным именем снова доступен.

  `set symbol-reloading off`
  : Не заменять определения символов при встрече объектного файла с таким же именем более одного раза. Это состояние по умолчанию; если вы не работаете в системе, допускающей автоматическое пересобирание модулей, вы должны оставить `symbol-reloading` в состоянии `off`, так как иначе GDB может уничтожить символы

при сборке больших программ, которые могут содержать несколько модулей с одинаковым именем (из разных каталогов или библиотек).

`show symbol-reloading`
: Показать текущую установку `on` или `off`.

`set opaque-type-resolution on`
: Велит GDB производить разрешение неоднозначных типов. Неоднозначным является тип, описанный как указатель на `struct`, `class` или `union`—например, `struct MyType *`—то есть, используется в одном исходном файле, хотя полное описание `struct MyType` находится в другом исходном файле. По умолчаний установлено в `on`.

Изменения в установке этой подкомандой не вступят в силу, пока символы для этого файла не будут снова загружены.

`set opaque-type-resolution off`
: Указывает GDB не производить разрешение неоднозначных типов. В этом случае, тип выводится следующим образом:

```
{<no data fields>}
```

`show opaque-type-resolution`
: Показать, производится разрешение неоднозначных типов или нет.

`maint print symbols` *имя-файла*
`maint print psymbols` *имя-файла*
`maint print msymbols` *имя-файла*
: Записать дамп данных отладочных символов в файл *имя-файла*. Эти команды используются для отладки кода GDB, осуществляющего чтение символов. Включаются только символы с отладочными данными. Если вы используете ‘`maint print symbols`’, GDB включает все символы, для которых уже собрана полная информация: то есть *имя-файла* отражает символы только для тех файлов, символы которых GDB уже прочитал. Вы можете использовать команду `info sources`, чтобы определить, какие это файлы. Если же вы используете ‘`maint print psymbols`’, дамп показывает информацию о тех символах, о которых GDB имеет лишь частичную информацию, то есть символы определены в файлах, которые GDB только просмотрел, но не прочитал полностью. Наконец, ‘`maint print msymbols`’, выводит только минимальную информацию о символах, требуемую для каждого объектного файла, из которых GDB прочитал какие-либо символы. См. Раздел 12.1 [Команды для задания файлов], с. 105, для обсуждения как GDB считывает символы (в описании `symbol-file`).

11 Изменение выполнения

Если вы думаете, что нашли ошибку в своей программе, вы можете захотеть выяснить наверняка, приведет ли исправление кажущейся ошибки к правильным результатам в остальной части программы. Вы можете получить ответ экспериментируя, используя средства GDB для изменения выполнения программы.

Например, вы можете сохранить новые значения в переменных или ячейках памяти, подать своей программе сигнал, перезапустить ее с другого адреса или даже преждевременно вернуться из функции.

11.1 Присваивание значений переменным

Для изменения значения переменной, вычислите выражение присваивания. См. Раздел 8.1 [Выражения], с. 61. Например,

```
print x=4
```

сохраняет значение 4 в переменной `x` и затем выводит значение данного выражения (которое равно 4). См. Глава 9 [Использование GDB с различными языками программирования], с. 77, для получения большей информации об операторах в поддерживаемых языках.

Если вы не хотите видеть значение присваивания, используйте команду `set` вместо `print`. Команда `set` аналогична команде `print` за исключением того, что значение выражения не выводится и не заносится в историю значений (см. Раздел 8.8 [История значений], с. 72). Выражение вычисляется только ради его действия.

Если начало строки параметров команды `set` выглядит идентично подкоманде `set`, используйте вместо нее команду `set variable`. Эта команда аналогична `set`, но не имеет подкоманд. Например, если в вашей программе есть переменная `width`, то вы получите ошибку, если попытаетесь установить новое значение просто с помощью ‘`set width=13`’, потому что GDB имеет команду `set width`:

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

Недопустимое выражение, это, конечно, ‘`=47`’. Для того чтобы действительно установить переменную программы `width`, используйте

```
(gdb) set var width=47
```

Так как команда `set` имеет много подкоманд, которые могут конфликтовать с именами переменных в программе, то хорошей практикой является использование команды `set variable` вместо просто `set`. Например, если ваша программа имеет переменную `g`, у вас возникнут проблемы, если вы попытаетесь установить новое значение с помощью ‘`set g=4`’, потому что GDB имеет команду `set gnutarget`, которая сокращается как `set g`:

```
(gdb) whatis g
type = double
(gdb) p g
$1 = 1
(gdb) set g=4
(gdb) p g
$2 = 1
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/smith/cc_progs/a.out
"/home/smith/cc_progs/a.out": can't open to read symbols:
                                 Invalid bfd target.
(gdb) show g
The current BFD target is "=4".
```

Переменная программы `g` не изменилась, и вы незаметно установили `gnutarget` в неверное значение. Для установки значения переменной `g`, используйте

```
(gdb) set var g=4
```

GDB допускает больше неявных преобразований в присваиваниях, чем Си; вы можете свободно сохранить целое значение в переменной-указателе и наоборот, преобразовать любую структуру к любой другой, которая имеет ту же длину или короче.

Для сохранения значений в произвольных местах памяти, используйте конструкцию '`{...}`' для создания значения определенного типа по определенному адресу памяти (см. Раздел 8.1 [Выражения], с. 61). Например, `{int}0x83040` ссылается на ячейку памяти `0x83040` как на целое (что предполагает соответствующий размер и представление в памяти), и

```
set {int}0x83040 = 4
```

записывает в эту ячейку памяти значение 4.

11.2 Продолжение исполнения с другого адреса

Обычно, когда вы продолжаете выполнение программы, вы делаете это с того места, где она остановилась, командой `continue`. Вместо этого, вы можете продолжить выполнение с любого выбранного адреса при помощи следующих команд:

`jump` *указ-стр*

Возобновить выполнение со строки *указ-стр*. Если там есть точка останова, выполнение немедленно прекращается. См. Раздел 7.1 [Вывод строк исходного текста], с. 55, для описания различных форм *указ-стр*. Использование команды `tbreak` вместе с `jump` является обычной практикой. См. Раздел 5.1.1 [Установка точек останова], с. 32.

Команда `jump` не изменяет ни текущий кадр стека, ни указатель стека, ни содержимое каких-либо ячеек памяти или регистров, кроме счетчика программы. Если строка *указ-стр* находится вне выполняющейся в настоящее время функции, результаты могут быть странными, если эти функции используют аргументы или локальные переменные разных типов. По этой причине, команда `jump` запрашивает подтверждение, если указанная строка не находится в функции, выполняющейся в настоящее время. Однако, даже странные результаты предсказуемы, если вы хорошо знакомы с машинным кодом вашей программы.

`jump *`*адрес*
: Возобновить выполнение с инструкции, находящейся по адресу *адрес*.

На многих системах, вы можете достичь такого же результата, как и с командой `jump`, сохранением нового значения в регистр `$pc`. Отличие в заключается том, что это не начинает выполнение вашей программы, а лишь изменяет адрес, с которого *будет* выполняться программа, когда вы продолжите выполнение. Например,

```
set $pc = 0x485
```

выполняет следующую команду `continue` или команду пошагового выполнения с адреса `0x485`, а не с того адреса, где ваша программа остановилась. См. Раздел 5.2 [Продолжение и выполнение по шагам], с. 44.

Наиболее общий случай использования команды `jump` состоит в возврате к выполнению части программы, возможно с большим количеством установленных точек останова, которая уже выполнилась, для того чтобы исследовать выполнение более детально.

11.3 Подача сигнала вашей программе

`signal` *сигнал*
: Возобновить выполнение с места остановки вашей программы, но немедленно подать ей сигнал *сигнал*. *Сигнал* может быть именем или номером сигнала. Например, во многих системах `signal 2` и `signal SIGINT`—два способа подать сигнал прерывания.

  Наоборот, если *сигнал* является нулем, выполнение продолжается без подачи сигнала. Это полезно, если ваша программа остановилась из-за сигнала и в обычном случае увидит его при возобновлении выполнения командой `continue`; ‘`signal 0`’ продолжит выполнение без сигнала.

  `signal` не повторяется, когда вы нажимаете ⟨RET⟩ второй раз после выполнения команды.

Вызов команды `signal` отличается от вызова утилиты `kill` из оболочки. Подача сигнала посредством `kill` заставляет GDB решать, что делать с сигналом, в зависимости от таблиц обработки сигналов (см. Раздел 5.3 [Сигналы], с. 46). Команда `signal` передает сигнал непосредственно вашей программе.

11.4 Возврат из функции

`return`
`return` *выражение*
: Вы можете отменить выполнение вызова функции с помощью команды `return`. Если вы задаете параметр *выражение*, его значение используется в качестве возвращаемого значения.

Когда вы используете `return`, GDB уничтожает выбранный кадр стека (и все кадры внутри него). Вы можете считать это преждевременным возвратом из уничтоженного кадра. Если вы хотите указать возвращаемое значение, задайте его в качестве аргумента к `return`.

Это выталкивает выбранный кадр стека (см. Раздел 6.3 [Выбор кадра стека], с. 52) и все другие кадры внутри него, оставляя самым внутренним кадр, из которого произошел вызов. Этот кадр становится выбранным. Указанное значение сохраняется в регистрах, используемых для возвращаемых функцией значений.

Команда `return` не возобновляет выполнение; она оставляет программу остановленной в том состоянии, в котором бы она была сразу после возврата из функции. Напротив,

команда `finish` (см. Раздел 5.2 [Продолжение и выполнение по шагам], с. 44) возобновляет выполнение до естественного возврата из выбранного кадра стека.

11.5 Вызов функций программы

`call` *выраж*
: Вычислить выражение *выраж* без отображения пустых (`void`) возвращенных значений.

Вы можете использовать этот вариант команды `print`, если хотите выполнить функцию из вашей программы, не засоряя вывод пустыми возвращенными значениями. Если результат не пустой, он выводится и сохраняется в истории значений.

Для A29K, устанавливаемая пользователем переменная `call_scratch_address` задает положение рабочей области, которая будет использоваться, когда GDB вызывает функцию на целевой машине. Это необходимо, так как обычный метод размещения рабочей области в стеке не работает в системах с раздельными областями команд и данных.

11.6 Внесение изменений в программу

По умолчанию, GDB открывает файл, содержащий исполняемый код вашей программы (или файл дампа памяти), в режиме только для чтения. Это предотвращает случайные изменения машинного кода; но это также предотвращает и преднамеренное исправление двоичного файла вашей программы.

Если вы хотите иметь возможность исправлять двоичный код, вы можете указать это явно с помощью команды `set write`. Например, вы можете захотеть установить внутренние флаги отладки или даже сделать аварийные исправления.

`set write on`
`set write off`
: Если вы установите ‘`set write on`’, GDB открывает исполняемые файлы и файлы дампов памяти в режиме для чтения и записи; если вы укажете ‘`set write off`’ (устанавливается по умолчанию), GDB открывает их в режиме только для чтения.

  Если вы уже загрузили файл, то после установки `set write` вам необходимо загрузить его снова (используя команды `exec-file` или `core-file`), чтобы новые установки вступили в силу.

`show write`
: Показать, открыты исполняемые файлы и файлы дампов памяти для записи или нет.

12 Файлы GDB

GDB должен знать имя файла программы, которая будет отлаживаться, чтобы прочитать его таблицу символов и чтобы запустить его. Для отладки дампа памяти от предыдущего выполнения, вы также должны сообщить GDB имя этого файла.

12.1 Команды для задания файлов

Вы можете указать имена исполняемого файла и файла дампа памяти. Обычно это делается во время вызова GDB, используя параметры с командами запуска GDB (см. Глава 2 [Вход и выход из GDB], с. 9).

Иногда во время сеанса GDB необходимо перейти к другому файлу. Или вы можете запустить GDB и забыть указать файл, который хотите использовать. В этих ситуациях полезны команды GDB для задания новых файлов.

`file` *имя-файла*
: Использовать *имя-файла* в качестве программы для отладки. Из нее читаются символы и содержание неизменяемой памяти. Также она является программой, которая выполняется при использовании команды `run`. Если вы не укажете каталог, и файл не будет найден в рабочем катологе GDB, он использует переменную среды `PATH` в качестве списка каталогов для поиска, точно так же, как это делает оболочка, когда ищет программу для выполнения. Используя команду `path`, вы можете изменить значение этой переменной как для GDB, так и для вашей программы.

  В системах с отображаемыми в память файлами, информация из таблицы символов для *имя-файла* может храниться во вспомогательном файле '*имя-файла*`.syms`'. Если это так, GDB осуществляет отображение таблицы символов из '*имя-файла*`.syms`', запускаясь намного быстрее. Смотрите описания ключей файлов '`-mapped`' и '`-readnow`' (доступных с командной строки и в командах `file`, `symbol-file` или `add-symbol-file`, описанных ниже), для получения большей информации.

`file`
: `file` без параметров велит GDB уничтожить любую имеющуюся информацию как об исполняемом файле, так и о таблице символов.

`exec-file` [*имя-файла*]
: Указывает, что программа, которая должна быть выполнена (но не таблица символов), находится в *имя-файла*. Если необходимо, GDB ищет вашу программу с помощью переменной среды `PATH`. Отсутствие *имя-файла* означает, что необходимо уничтожить информацию о выполняемом файле.

`symbol-file` [*имя-файла*]
: Читать информацию таблицы символов из файла *имя-файла*. При необходимости производится поиск с помощью переменной среды `PATH`. Для получения таблицы символов и исполняемой программы из одного и того же файла, используйте команду `file`.

  `symbol-file` без параметров сбрасывает информацию GDB о таблице символов вашей программы.

  Команда `symbol-file` велит GDB забыть содержимое своих вспомогательных переменных, историю значений и все точки останова и выражения автоматического отображения, так как они могут содержать указатели на внутренние данные, хранящие символы и типы данных, которые являются частью данных старой таблицы символов, уничтоженной внутри GDB.

`symbol-file` не повторяется, если вы снова нажимаете ⟨RET⟩ после первого выполнения.

Когда GDB сконфигурирован для определенной среды, он распознает отладочную информацию в том формате, который обычно генерируется для этой среды; вы можете использовать или компилятор GNU, или другие компиляторы, которые придерживаются местных соглашений. Наилучшие результаты обычно достигаются с помощью компилятора GNU; например, используя `gcc`, вы можете создавать отладочную информацию для оптимизированного кода.

Для большинства типов объектных файлов, за исключением старых систем SVR3, использующих COFF, команда `symbol-file` обычно не считывает таблицу символов сразу целиком. Вместо этого, она быстро сканирует ее для определения, какие исходные файлы и символы в ней присутствуют. Детали читаются позже, по одному исходному файлу за раз, по мере необходимости.

Такая стратегия чтения в две стадии используется для того, чтобы GDB вызывался быстрее. За исключением редких пауз, чтение деталей таблицы символов для конкретного исходного файла в большинстве случаев практически незаметно. (Команда `set verbose` позволяет при желании превратить эти паузы в сообщения. См. Раздел 15.6 [Необязательные предупреждения и сообщения], с. 154.)

Мы еще не реализовали двухступенчатую стратегию чтения для COFF. Когда таблица символов сохранена в формате COFF, `symbol-file` считывает данные таблицы символов сразу полностью. Заметьте, что “stabs-in-COFF” все же реализует двухступенчатую стратегию, так как отладочная информация реально хранится в формате stabs.

`symbol-file` *имя-файла* [`-readnow`] [`-mapped`]
`file` *имя-файла* [`-readnow`] [`-mapped`]

Если вы хотите быть уверены, что у GDB есть вся таблица символов целиком, вы можете отменить двухступенчатую стратегию чтения таблицы символов, используя параметр ‘`-readnow`’ с любой командой, загружающей информацию таблицы символов.

Если отображаемые в память файлы доступны в вашей системе через системный вызов `mmap`, вы можете использовать другой параметр, ‘`-mapped`’, чтобы GDB записывал символы для вашей программы в файл многократного использования. Последующие сеансы отладки GDB отображают информацию о символах из этого вспомогательного файла (если программа не изменилась), вместо того, чтобы тратить время на чтение таблицы символов из исполняемой программы. Использование параметра ‘`-mapped`’ производит такой же эффект, как вызов GDB с ключем командной строки ‘`-mapped`’.

Вы можете использовать оба параметра вместе, чтобы быть уверенным, что вспомогательный файл символов содержит всю информацию о символах вашей программы.

Вспомогательный файл символов для программы *myprog* называется ‘*myprog*`.syms`’. Если этот файл существует (и создан позже, чем соответствующая исполняемая программа), GDB всегда пытается использовать его при отладке *myprog*; не требуется никаких специальных ключей или команд.

Файл ‘`.syms`’ является специфичным для рабочей машины, на которой вы вызываете GDB. Он содержит точный образ внутренней таблицы символов GDB. Он не может быть использован одновременно на разных рабочих платформах.

`core-file` [*имя-файла*]

Определяет местонахождение файла дампа памяти, который будет использован как “содержимое памяти”. Обычно файлы дампов памяти содержат только

некоторые части адресного пространства процесса, создавшего их; GDB может обращаться к исполняемому файлу за другими частями.

`core-file` без параметра указывает, что файл дампа памяти использоваться не должен.

Обратите внимание, что файл дампа памяти игнорируется, если в данное время ваша программа выполняется под управлением GDB. Так что если вы выполняли программу, и желаете вместо этого отладить файл дампа, вы должны убить подпроцесс, в котором выполняется ваша программа. Для этого используйте команду `kill` (см. Раздел 4.8 [Уничтожение дочернего процесса], с. 25).

`add-symbol-file` *имя-файла адрес*
`add-symbol-file` *имя-файла адрес* [`-readnow`] [`-mapped`]
`add-symbol-file` *имя-файла* `-s`*раздел адрес*

Команда `add-symbol-file` считывает дополнительную информацию таблицы символов из файла *имя-файла*. Вы должны использовать эту команду, если файл *имя-файла* был динамически загружен (другими средствами) в выполняющуюся программу. *Адрес* должен быть адресом памяти, по которому был загружен файл; GDB сам его определить не может. Вы можете указать дополнительно произвольное количество пар '-s *раздел адрес*', чтобы явно указать имя раздела и базовый адрес для него. Вы можете указать произвольный *адрес* как выражение.

Таблица символов из файла *имя-файла* добавляется к таблице, изначально считанной по команде `symbol-file`. Вы можете использовать команду `add-symbol-file` произвольное число раз; прочитанные таким образом символьные данные добавляются к старым. Чтобы уничтожить все старые данные, используйте команду `symbol-file` без аргументов.

Команда `add-symbol-file` не повторяется, если вы нажимаете ⟨RET⟩ после ее использования.

Чтобы изменить способ обработки GDB таблицы символов для *имя-файла*, вы можете использовать параметры '`-mapped`' и '`-readnow`' так же, как и с командой `symbol-file`.

`add-shared-symbol-file`

Команда `add-shared-symbol-file` может быть использована только для Motorola 88k в операционной системе Harris CXUX. GDB ищет разделяемые библиотеки автоматически, однако, если он не находит ваших, вы можете выполнить `add-shared-symbol-file`. Эта команда не имеет аргументов.

`section` Команда `section` изменяет базовый адрес раздела *раздел* выполняемого файла в *адрес*. Это может быть использовано, если выполняемый файл не содержит адресов разделов, (что имеет место для формата a.out), или когда адреса, указанные в самом файле, неверны. Каждый раздел должен изменяться отдельно. Команда `info files`, описанная ниже, перечисляет все разделы и их адреса.

`info files`
`info target`

Команды `info files` и `info target` являются синонимами; они обе выводят текущую цель (см. Глава 13 [Определение отладочной цели], с. 111), включая имена выполняемого файла и файла дампа памяти, используемых GDB, и файлов, из которых были загружены символы. Команда `help target` выводит все возможные цели, а не только текущую.

Все команды для задания файлов допускают в качестве аргументов как абсолютные, так и относительные имена файлов. GDB всегда преобразовывает имя файла к абсолютному и запоминает его в таком виде.

GDB поддерживает разделяемые библиотеки на HP-UX, SunOS, SVr4, Irix 5 и IBM RS/6000.

Когда вы даете команду `run`, или когда исследуете файл дампа памяти, GDB автоматически загружает определения символов из разделяемых библиотек. (Однако, если вы не отлаживаете файл дампа, GDB не понимает ссылки на функции из разделяемой библиотеки до того, как вы выполните команду `run`.)

На HP-UX, если программа загружает разделяемую библиотеку явно, GDB автоматически загружает символы в момент вызова `shl_load`.

`info share`
`info sharedlibrary`
: Вывести имена разделяемых библиотек, загруженных в данный момент.

`sharedlibrary` *рег-выр*
`share` *рег-выр*
: Загрузить символы разделяемых библиотек объектов для файлов, удовлетворяющих регулярному выражению Unix. Также как и для автоматически загруженных файлов, это загружает только разделяемые библиотеки, требуемые вашей программой для файла дампа памяти или после ввода `run`. Если *рег-выр* опущено, загружаются все разделяемые библиотеки, требуемые вашей программой.

В системах HP-UX, GDB сам определяет загрузку разделяемой библиотеки и автоматически считывает символы из нее, до некоторого изначально установленного порогового значения, которое вы можете при желании изменить.

После этого порогового значения, символы из разделяемых библиотек должны загружаться явно. Для загрузки этих символов, используйте команду `sharedlibrary` *имя-файла*. Базовый адрес разделяемой библиотеки определяется GDB автоматически и вы не должны его задавать.

Для отображения или установки порогового значения, используйте следующие команды:

`set auto-solib-add` *порог*
: Устанавливает размер порога автоматической загрузки в мегабайтах. Если *порог* ненулевой, символы из всех библиотек разделяемых объектов будут загружаться автоматически, когда программа начинает выполнение или когда динамический компоновщик информирует GDB о том, что была загружена новая библиотека, до тех пор, пока таблица символов программы и библиотек не превысит этот порог. В противном случае, символы должны загружаться вручную, при помощи команды `sharedlibrary`. По умолчанию, порог равен 100 мегабайтам.

`show auto-solib-add`
: Отобразить величину текущего порога автоматической загрузки в мегабайтах.

12.2 Ошибки чтения файлов с символами

При чтении файла символов, GDB иногда сталкивается с такими проблемами, как типы символов, которые он не распознает, или известные ошибки вывода компилятора. По умолчанию, GDB не сообщает вам о таких проблемах, так как они сравнительно общие и прежде всего представляют интерес для людей, занимающихся отладкой компиляторов. Если вам интересна информация о плохо созданных таблицах символов, вы можете запросить GDB печатать только одно сообщение по каждому типу проблем, независимо от того, сколько раз проблема появляется; или вы можете попросить GDB напечатать больше

сообщений, чтобы увидеть, сколько раз проблема встречалась, командой `set complaints` (см. Раздел 15.6 [Необязательные предупреждения и сообщения], с. 154).

Печатаемые сообщения и их значения, включают:

`inner block not inside outer block in` *символ*
: Информация о символах показывает, где области символов начинаются и заканчиваются (например, в начале функции или блока операторов). Эта ошибка указывает на то, что внутренний блок видимости не содержится целиком во внешнем.

  GDB обходит проблему, рассматривая внутренний блок так, как если бы он имел такую же область видимости, что и внешний. Если внешний блок не является функцией, в данном сообщении *символ* может быть показан как "`(don't known)`".

`block at` *адрес* `out of order`
: Символьная информация для блоков символьных областей должна появляться в порядке увеличения адресов. Данная ошибка указывает, что это не так.

  GDB не решает этой проблемы, и у него возникают трудности при определении местоположения символов в исходном файле, символы которого он считывает. (Вы часто можете определить имя поврежденного исходного файла, указав `set verbose on`. См. Раздел 15.6 [Необязательные предупреждения и сообщения], с. 154.)

`bad block start address patched`
: Символьная информация для блоков символьных областей имеет меньший начальный адрес, чем у предшествующей строки исходного текста. Известно, что это происходит в компиляторе Си SunOS 4.1.1 (и более ранних версиях).

  GDB обходит проблему, обрабатывая блок символьной области как начинающийся с предыдущей исходной строки.

`bad string table offset in symbol` *n*
: Символ с номером *n* содержит указатель на таблицу строк, который превосходит размер таблицы.

  GDB обходит проблему, считая, что символ имеет имя `foo`, что может вызвать другие проблемы, если много символов заканчиваются этим именем.

`unknown symbol type 0x`*nn*
: Символьная информация содержит новые типы данных, которые GDB еще не знает, как считывать. `0x`*nn*—это тип символа неверно истолкованной информации, в шестнадцатеричном виде.

  GDB обходит ошибку, игнорируя эту символьную информацию. Это обычно позволяет вам отладживать программу, хотя некоторые символы и недоступны. Если вы столкнетесь с такой проблемой и желаете ее отладить, вы можете отладить `gdb` с помощью него же, установив точку останова на `complain`, затем дойти до функции `read_dbx_symtab` и исследовать `*bufp`, чтобы увидеть символ.

`stub type has NULL name`
: GDB не может найти полное определение для структуры или класса.

`const/volatile indicator missing (ok if using g++ v1.x), got...`
: В символьной информации для функции-члена Си++ пропущена некоторая информация, которую последние версии компилятора должны для нее выводить.

`info mismatch between compiler and debugger`
: GDB не может разобрать спецификации типа, выведенной компилятором.

13 Определение отладочной цели

Цель—это среда выполнения, занятая вашей программой.

Часто GDB выполняется в той же рабочей среде, что и ваша программа; в этом случае, отладочная цель задается неявно в момент использования команд `file` или `core`. Когда вам нужна большая гибкость—например, выполнение GDB на другой машине, или управление автономной системой через последовательный порт или системой реального времени через соединение TCP/IP—вы можете использовать команду `target` для определения цели одного из типов, сконфигурированных для GDB (см. Раздел 13.2 [Команды для управления целями], с. 111).

13.1 Активные цели

Существует три класса целей: процессы, файлы дампов памяти и выполняемые файлы. GDB может обрабатывать одновременно до трех активных целей, по одной в каждом классе. Это позволяет вам (например) запустить процесс и проверять его действия, не прерывая вашу работу над файлом дампа.

Например, если вы выполняете ‘`gdb a.out`’, то исполняемый файл `a.out` является единственной активной целью. Если вы назначите также файл дампа—возможно от предыдущего выполнения, завершившегося ошибкой и создавшего дамп— тогда GDB имеет две активные цели и использует их вместе, просматривая сначала файл дампа, а затем исполняемый файл, для выполнения запросов к адресам памяти. (Обычно эти два класса целей дополняют друг друга, так как файлы дампа памяти содержит только память программы, доступную для чтения и записи (переменные и тому подобное), и машинное состояние, в то время как исполняемый файлы содержат только текст программы и инициализированные данные.)

Когда вы вводите `run`, ваш исполняемый файл становится также активным целевым процессом. Когда целевой процесс активен, все команды GDB, запрашивающие адреса памяти, относятся к этой цели; адреса в активной цели файла дампа или выполняемого файла неизвестны, пока активен целевой процесс.

Используйте команды `core-file` и `exec-file` для выбора новой цели файла дампа памяти или выполняемого файла (см. Раздел 12.1 [Команды для задания файлов], с. 105). Для определения в качестве цели процесса, который уже выполняется, используйте команду `attach` (см. Раздел 4.7 [Отладка запущенного ранее процесса], с. 25).

13.2 Команды для управления целями

`target` *тип параметры*

Соединяет рабочую среду GDB с целевой машиной или процессом. Целью обычно является протокол для взаимодействия со средствами отладки. Параметр *тип* используется, чтобы определить тип или протокол целевой машины.

Дальнейшие *параметры* интерпретируются целевым протоколом, но обычно включают такие вещи, как имена устройств или имена рабочих машин, с которыми осуществляется связь, номера процессов и скорости в бодах.

Команда `target` не повторяется при повторном нажатии ⟨RET⟩ после ее выполнения.

`help target`

Отображает имена всех доступных целей. Чтобы отобразить выбранные в данный момент цели, используйте либо `info target`, либо `info files` (см. Раздел 12.1 [Команды для задания файлов], с. 105).

`help target` *имя*
: Описывает определенную цель, включая любые параметры, необходимые для ее выбора.

`set gnutarget` *арг*
: GDB использует свою собственную библиотеку BFD[1] для чтения ваших файлов. GDB знает, читает ли он *выполняемый файл*, *файл дампа памяти* или *объектный (.o)* файл; однако вы можете определить формат файла командой `set gnutarget`. В отличие от большинства команд `target`, с `gnutarget`, команда `target` относится к программе, а не к машине.

  > *Предупреждение:* Для определения формата файла посредством `set gnutarget`, вы должны знать фактическое имя BFD.

  См. Раздел 12.1 [Команды для задания файлов], с. 105.

`show gnutarget`
: Используйте команду `show gnutarget` для отображения, какого формата файлы `gnutarget` установлен считывать. Если вы не установили `gnutarget`, GDB определит формат для каждого файла автоматически, и `show gnutarget` выведет ‘`The current BFD target is "auto"`’.

Ниже приведены некоторые наиболее распространенные цели (доступные или нет, в зависимоси от конфигурации GDB):

`target exec` *программа*
: Выполняемый файл. ‘`target exec` *программа*’ то же самое, что и ‘`exec-file` *программа*’.

`target core` *имя-файла*
: Файл дампа памяти. ‘`target core` *имя-файла*’ то же самое, что и ‘`core-file` *имя-файла*’.

`target remote` *устр*
: Удаленная последовательная цель является уникальным для GDB протоколом. Параметр *устр* определяет, какое последовательное устройство использовать для соединения (например, ‘`/dev/ttya`’). См. Раздел 13.4 [Удаленная отладка], с. 113. `target remote` поддерживает команду `load`. Это полезно, только если вы можете получить заглушку для целевой системы каким-нибудь другим способом и можете разместить ее в памяти, где она не будет затерта загрузкой.

`target sim`
: Встроенный эмулятор ЦП. GDB включает эмуляторы для большинства архитектур. Вообще,

```
target sim
load
run
```

  работает; однако, вы не можете предположить, что доступны определенное отображение памяти, драйверы устройств, или даже основные функции ввода-вывода, хотя некоторые эмуляторы действительно предоставляют это. Для информации о деталях эмуляторов для конкретного процессора, смотрите соответствующий Раздел 14.3 [Встроенные процессоры], с. 136.

Некоторые конфигурации могут также включать такие цели:

`target nrom` *устр*
: Эмулятор NetROM ROM. Эта цель поддерживает только загрузку.

[1] от ‘Binary File Descriptor’ (библиотека описания двоичных файлов). *(Прим. переводчика)*

Для различных конфигураций GDB доступны различные цели; ваша конфигурация может иметь больше или меньше целей.

Многие удаленные цели требуют, чтобы вы загрузили код выполняемого файла, после того как вы успешно установили соединение.

`load` *имя-файла*
: В зависимости от того, какие возможности удаленной отладки сконфигурированы в GDB, может быть доступна команда `load`. Если она существует, ее задачей является сделать *имя-файла* (выполняемый файл) доступным для отладки на удаленной системе—например, путем загрузки или динамической сборки. `load` также записывает таблицу символов *имя-файла* в GDB, как команда `add-symbol-file`.

  Если ваш GDB не имеет команды `load`, попытка выполнить ее выдает сообщение об ошибке "`You can't do that when your target is ...`".

  Файл загружается по адресу, указанному в выполняемом файле. Для некоторых форматов объектных файлов, вы можете задать адрес загрузки при сборке программы; для других форматов, таких как a.out, формат объектного файла задает фиксированный адрес.

  `load` не повторяется, если вы нажимаете ⟨RET⟩ снова после ее использования.

13.3 Выбор целевого порядка байтов

Некоторые типы процессоров, такие как MIPS, PowerPC и Hitachi SH, предоставляют возможность выполнения либо с порядком байтов от старшего, либо с порядком байтов от младшего. Обычно, выполняемый файл или символы содержат информацию для определения используемого порядка байтов, и вам не нужно об этом заботиться. Однако, иногда вам все же может пригодиться вручную изменить порядок байтов процессора, определенный GDB.

`set endian big`
: Велит GDB считать, что целевой порядок байтов от старшего.

`set endian little`
: Велит GDB считать, что целевой порядок байтов от младшего.

`set endian auto`
: Велит GDB использовать порядок байтов, указанный в выполняемом файле.

`show endian`
: Отображает текущую установку GDB для целевого порядка байтов.

Заметьте, что эти команды управляют только интерпретацией символьных данных в рабочей системе, и они совершенно не оказывают действия на целевую систему.

13.4 Удаленная отладка

Если вы пытаетесь отлаживать программу, выполняющуюся на машине, которая не может запустить GDB обычным способом, часто бывает полезна удаленная отладка. Например, вы можете использовать удаленную отладку для ядра операционной системы или для малой системы, которая не имеет достаточно мощной операционной системы общего назначения для вызова отладчика со всеми возможностями.

Некоторые конфигурации GDB имеют специальный последовательный или TCP/IP интерфейсы для того, чтобы это работало с конкретными отладочными целями. Кроме того, GDB распространяется с общим последовательным протоколом (уникальным для GDB,

но не для конкретной целевой системы), который вы можете использовать, если пишете удаленные заглушки—код, выполняемый в удаленной системе для связи с GDB.

В вашей конфигурации GDB могут быть доступны другие удаленные цели; используете `help target`, чтобы их перечислить.

13.4.1 Удаленный последовательный протокол GDB

Для отладки программы, выполняемой на другой машине (отладочной *целевой* машине), вы сперва должны создать все обычные предпосылки для самостоятельного выполнения программы. Например, для программы на Си вам нужны:

1. Процедура запуска для установки среды выполнения Си; она обычно имеет имя типа '`crt0`'. Процедура запуска может быть обеспечена вашими аппаратными средствами, или вы должны написать свою собственную.
2. Библиотека подпрограмм Си для поддержки вызовов подпрограмм вашей программы, особенно для управления вводом и выводом.
3. Способ установки вашей программы на другую машину—например, программа загрузки. Такие программы часто предоставляются поставщиками аппаратных средств, но вам может потребоваться написать вашу собственную, пользуясь документацией к аппаратному обеспечению.

Следующим шагом будет принятие мер по использованию вашей программой последовательного порта для связи с машиной, где выполняется GDB (*рабочей* машиной). В общих чертах, схема выглядит следующим образом:

На рабочей машине
: GDB уже понимает, как использовать этот протокол; после установки всего остального, вы можете просто использовать команду '`target remote`' (см. Глава 13 [Определение отладочной цели], с. 111).

На целевой машине
: вы должны скомпоновать вместе с вашей программой несколько подпрограмм специального назначения, которые реализуют удаленный последовательный протокол GDB. Файл, содержащий эти подпрограммы, называется *отладочной заглушкой*.

  На некоторых удаленных целях, вы можете использовать вспомогательную программу `gdbserver` вместо компоновки заглушки вместе с вашей программой. См. Раздел 13.4.1.5 [Использование программы gdbserver], с. 130, для детального изучения.

Отладочная заглушка специфична для архитектуры удаленной машины; например, используйте '`sparc-stub.c`' для отладки программ на машинах SPARC.

Следующие работающие удаленные заглушки распространяются вместе с GDB:

`i386-stub.c`
: Для Intel 386 и совместимых архитектур.

`m68k-stub.c`
: Для архитектур Motorola 680x0.

`sh-stub.c`
: Для архитектур Hitachi SH.

`sparc-stub.c`
: Для архитектур SPARC.

`sparcl-stub.c`
: Для архитектур Fujitsu SPACRLITE.

Файл '`README`' в поставке GDB может содержать другие недавно добавленные заглушки.

13.4.1.1 Что заглушка может сделать для вас

Отладочная заглушка для вашей архитектуры содержит следующие три подпрограммы:

`set_debug_traps`
: Когда ваша программа останавливается, эта подпрограмма организует выполнение `handle_exception`. Вы должны явно вызвать эту подпрограмму в начале вашей программы.

`handle_exception`
: Это главная рабочая лошадка, но ваша программа никогда не вызывает ее явно—установочный код организует запуск `handle_exception`, когда вызывается ловушка.

  `handle_exception` получает управление, когда ваша программа останавливается во время выполнения (например, в точке останова), и организует связь с GDB на рабочей машине. Именно здесь реализуется протокол связи; `handle_exception` действует как представитель GDB на целевой машине. Сперва она посылает суммарную информацию о состоянии вашей программы, затем продолжает выполняться, извлекая и передавая любую информацию, требующуюся GDB, пока вы не выполните команду GDB, возобновляющую выполнение вашей программы; в этом месте `handle_exception` возвращает управление вашему коду на целевой машине.

`breakpoint`
: Используйте эту вспомогательную подпрограмму для установки в вашей программе точек останова. В зависимости от конкретной ситуации, это может быть единственным способом для GDB получить управление. Например, если ваша целевая машина имеет некую клавишу прерывания, вам не нужно вызывать эту подпрограмму; нажатие клавиши прерывания передаст управление `handle_exception`—в действительности, GDB. На некоторых машинах простое получение символов на последовательный порт может также вызвать ловушку; опять, в этой ситуации вам не нужно вызывать `breakpoint` из вашей программы—простое выполнение ‘`target remote`’ из рабочего сеанса GDB передаст управление.

  Вызывайте `breakpoint`, если ни одно из этих предположений не верно, или вы просто хотите быть уверенным, что ваша программа остановится в предопределенной точке от начала вашего сеанса отладки.

13.4.1.2 Что вы должны сделать для заглушки

Отладочные заглушки, поставляемые с GDB, ориентированы на микропроцессоры определенной архитектуры, но они не имеют информации об остальной части вашей целевой отладочной машины.

В первую очередь, вам нужно сообщить заглушке, как связаться с последовательным портом.

`int getDebugChar()`
: Напишите эту подпрограмму для чтения одного символа из последовательного порта. Она может быть идентична `getchar` для вашей целевой системы; разные имена используются, чтобы позволить вам их различать, если вы этого хотите.

`void putDebugChar(int)`
: Напишите эту подпрограмму для записи одного символа в последовательный порт. Она может быть идентична `putchar` для вашей целевой системы; разные имена используются, чтобы позволить вам их различать, если вы этого хотите.

Если вы хотите, чтобы GDB мог остановить вашу программу во время ее выполнения, вам нужно использовать управляемый прерываниями последовательный драйвер и настроить его для остановки при получении `^C` (‘`\003`’, символ control-C). Это тот символ, который GDB использует для указания удаленной системе остановиться.

Указание отладочной цели вернуть GDB правильный статус, вероятно, требует изменений стандартной заглушки; один быстрый и неаккуратный способ состоит в выполнении лишь инструкции точки останова (“неаккуратность” состоит в том, что GDB выдает `SIGTRAP` вместо `SIGINT`).

Вот другие процедуры, которые вы должны обеспечить:

`void exceptionHandler (int` *номер-исключения*`, void *`*адрес-исключения*`)`

Напишите эту функцию для установки *адреса-исключения* в таблицы обработки исключительных ситуаций. Вам нужно сделать это, потому что у заглушки нет способа узнать, как устроены таблицы обработки исключений в вашей целевой системе (например, процессорная таблица может быть в ПЗУ, и содержать элементы, указывающие на таблицу в ОЗУ). *Номер-исключения*—это номер исключительной ситуации, которая должна быть изменена; его значение зависит от архитектуры (например, различные номера могут представлять деление на ноль, доступ с нарушением выравнивания, и так далее). Когда это исключение возникает, управление должно быть передано непосредственно *адресу-исключения*, и процессорное состояние (стек, регистры и так далее) должно быть таким же, как во время возникновения процессорного исключения. Так что если вы хотите использовать инструкцию перехода для достижения *адреса-исключения*, это должен быть простой переход, не переход к подпрограмме.

Для 386, *адрес-исключения* должен быть установлен как обработчик затвора вызова прерывания, чтобы во время его выполнения остальные прерывания маскировались. Он должен иметь уровень полномочий 0 (наибольшие полномочия). Заглушки SPARC и 68k могут маскировать прерывания самостоятельно без помощи `exceptionHandler`.

`void flush_i_cache()`

Только для SPARC и SPARCLITE. Напишите эту попрограмму для очистки кеша инструкций, если он есть, на вашей целевой машине. Если кеша инструкций нет, эта подпрограмма может ничего не делать.

На целевых машинах, имеющих кеш инструкций, GDB требует эту функцию, чтобы удостовериться, что состояние вашей программы стабильное.

Вы должны также удостовериться, что эта библиотечная процедура доступна:

`void *memset(void *, int, int)`

Это стандартная библиотечная функция `memset`, которая устанавливает область памяти в заданное значение. Если вы имеете одну из свободных версий `libc.a`, `memset` может быть найдена там; иначе вы должны или получить ее от изготовителя аппаратного обеспечения, или написать свою собственную.

Если вы не используете компилятор GNU Си, вам также могут понадобиться другие стандартные библиотечные подпрограммы; это меняется от одной заглушки к другой, но в общем, заглушки часто используют различные общие библиотечные подпрограммы, которые `gcc` генерирует как встроенный код.

13.4.1.3 Собираем все вместе

Вкратце, когда ваша программа готова к отладке, вы должны проделать следующие шаги.

1. Убедитесь, что вы определили поддерживающие процедуры низкого уровня (см. Раздел 13.4.1.2 [Что вы должны сделать для заглушки], с. 115):

   `getDebugChar`, `putDebugChar`,
   `flush_i_cache`, `memset`, `exceptionHandler`.

2. Вставьте следующие строки в начале вашей программы:

   ```
   set_debug_traps();
   breakpoint();
   ```

3. Для заглушки 680x0, вы должны предоставить переменную `exceptionHook`. Обычно вы используете просто:

   ```
   void (*exceptionHook)() = 0;
   ```

   если до вызова `set_debug_traps` вы установили ее для указания на функцию в вашей программе. Эта функция вызывается, когда GDB продолжает выполнение после останова на ловушке (например, ошибка шины). Функция, указанная `exceptionHook`, вызывается с одним параметром типа `int`, который является номером исключения.

4. Откомпилируйте и скомпонуйте вместе: вашу программу, отладочную заглушку GDB для вашей целевой архитектуры и подпрограммы поддержки.

5. Убедитесь, что у вас есть последовательное соединение между вашей целевой и рабочей машинами, и идентифицируйте последовательный порт на рабочей машине.

6. Загрузите вашу программу на целевую машину (или поместите ее туда любыми средствами, предоставляемыми производителем) и запустите ее.

7. Для начала удаленной отладки, запустите GDB на рабочей машине и укажите в качестве выполняемого файла программу, которая выполняется на удаленной машине. Это сообщает GDB, как найти символы и содержание неизменяемых областей вашей программы.

8. Установите связь, используя команду `target remote`. Ее аргументы определяют, как взаимодействовать с целевой машиной— либо через устройство, подключенное к последовательной линии, либо через порт TCP (обычно подключенный к терминальному серверу, который, в свою очередь, имеет последовательную линию до цели). Например, чтобы использовать последовательную линию, присоединенную к устройству '`/dev/ttyb`', выполните:

   ```
   target remote /dev/ttyb
   ```

   Чтобы использовать TCP-соединение, используйте аргумент в форме *машина*`:порт`. Например, для соединения с портом 2828 на терминальном сервере `manyfarms`:

   ```
   target remote manyfarms:2828
   ```

Теперь вы можете использовать все обычные команды для исследования и изменения данных, пошагового выполнения и продолжения исполнения удаленной программы.

Для возобновления выполнения удаленной программы и прекращения ее отладки, используйте команду `detach`.

Всякий раз, когда GDB ожидает удаленную программу, если вы вводите символ прерывания (часто *C-C*), GDB пытается остановить программу. Это может привести или не привести к успеху, частично в зависимости от аппаратных средств и последовательных драйверов, которые использует удаленная система. Если вы снова введете символ прерывания, GDB выведет такое приглашение:

```
Interrupted while waiting for the program.
Give up (and stop debugging it)?  (y or n)
```

Если вы введете *y*, GDB прекратит сеанс удаленной отладки. (Если вы решите позже, что хотите попытаться снова, вы можете вновь использовать `target remote`, чтобы соединиться еще раз.) Если вы введете *n*, GDB вернется к ожиданию.

13.4.1.4 Коммуникационный протокол

Файлы заглушек, поставляемые с GDB, реализуют коммуникационный протокол со стороны целевой машины, а со стороны GDB он реализуется в исходном файле GDB ‘`remote.c`’. Обычно вы можете просто позволить этим программам взаимодействовать, и не вдаваться в детали. (Если вы разрабатываете свой собственный файл заглушки, вы также можете игнорировать детали: начните с одного из существующих файлов заглушки. ‘`sparc-stub.c`’ организован наилучшим образом, и потому его легче всего читать.)

Однако, бывают случаи, когда вам необходимо что-нибудь знать о протоколе— например, если существует только один последовательный порт на вашей целевой машине, вы можете захотеть, чтобы ваша программа делала что-нибудь особенное, если она распознает предназначенный для GDB пакет.

В следующих примерах, ‘`<-`’ и ‘`->`’ используются для обозначения переданных и полученных данных соответственно.

Все команды и ответы GDB (не подтверждения), посылаются в виде *пакета*. *Пакет* представлен символом ‘`$`’, реальными *данными-пакета* завершающим символом ‘`#`’, за которым следуют две цифры *контрольной-суммы*:

`$`*данные-пакета*`#`*контрольная-сумма*

Двухциферная *контрольная-сумма* вычисляется как сумма по модулю 256 всех символов между начальным ‘`$`’ и конечным ‘`#`’ (восьмибитная беззнаковая контрольная сумма).

Разработчикам следует учесть, что до GDB версии 5.0 спецификация протокола также включала необязательный двухциферный *ид-последов*:

`$`*ид-последов*`:`*данные-пакета*`#`*контрольная-сумма*

Этот *ид-последов* добавлялся к подтверждению. GDB никогда не генерировал *ид-последов*. Заглушки, занимающиеся обработкой пакетов, добавленные в GDB начиная с версии 5.0, не должны принимать пакеты с *ид-последов*.

Когда или рабочая, или целевая машина получает пакет, первым ожидаемым ответом является подтверждение: или ‘`+`’ (для указания, что пакет получен корректно), или ‘`-`’ (чтобы запросить повторную передачу):

```
<- $данные-пакета#контрольная-сумма
-> +
```

Рабочая машина (GDB) посылает *команды*, а целевая (отладочная заглушка, включенная в вашу программу) посылает *ответ*. В случае *команд* пошагового выполнения и продолжения, ответ посылается только тогда, когда операция закончена (цель снова остановлена).

Данные-пакета состоят из последовательности знаков, за исключением ‘`#`’ и ‘`$`’ (для дополнительных исключений, смотрите пакет ‘`X`’).

Поля внутри пакета должны разделяться при помощи ‘`,`’, ‘`;`’ или ‘`:`’. Если не оговорено противное, все числа представлены в шестнадцатеричном виде без начальных нулей.

Разработчикам следует учесть, что до GDB версии 5.0, символ ‘`:`’ не мог появляться третим символом в пакете (так как потенциально это могло привести к конфликту с *ид-последов*).

Ответ *данные* может быть закодированным с помощью кодирования методом длины серий, чтобы сохранить место. ‘`*`’ означает, что следующий символ является ASCII-кодом, который означает количество повторений символа, предшествующего ‘`*`’. Кодировкой является `n+29`, что дает печатный знак для `n >=3` (когда кодировка переменной длины дает преимущества). Печатные знаки ‘`$`’, ‘`#`’, ‘`+`’, ‘`-`’, или с номерами, большими 126, использоваться не должны.

Некоторые удаленные системы использовали другой механизм кодировки с переменной длиной, иногда называемый cisco-кодировкой. За ‘*’ следуют две шестнадцатеричные цифры, обозначающие размер пакета.

Итак:

```
"0* "
```

означает то же, что и "0000".

При ошибке, ответ, возвращаемый для некоторых пакетов, включает двухсимвольный номер ошибки. Этот номер определен смутно.

Для любой *команды*, не поддерживаемой заглушкой, должен быть возвращен пустой ответ (‘$#00’). Таким образом, протокол можно расширять. Новые версии GDB могут определить, поддерживается ли пакет, основываясь на ответе.

От заглушки требуется поддержка *команд* ‘g’, ‘G’, ‘m’, ‘M’, ‘c’ и ‘s’. Все остальные *команды* являются необязательными.

Вот полный список всех определенных на данный момент *команд*, и соответствующих им ответов *данные*:

Пакет	Запрос	Описание
расширенные операции	`!`	Использовать расширенный удаленный протокол. Имеет постоянное действие—требует установки только один раз. Расширенный удаленный протокол поддерживает пакеты ‘`R`’.
	ответ ‘’	Заглушки, поддерживающие расширенный удаленный протокол, возвращают ‘’, что, к сожалению, совпадает с ответом, возвращаемым заглушками, которые не поддерживают расширения протокола.
последний сигнал	`?`	Указывает причину, по которой цель остановилась. Ответ такой же, как для пошагового выполнения и продолжения.
	ответ	смотрите ниже
зарезервировано	`a`	Зарезервировано для использования в будущем
установить аргументы программы **(зарезервировано)**	A*длина-арг*`,`*число-арг*`,`*арг*`,...`	В программу передается инициализированный массив ‘`argv[]`’. *Длина-арг* задает число байт в закодированном в шестнадцатеричный вид потоке байт *арг*. Смотрите ‘`gdbserver`’ для дополнительной информации.
	ответ `OK` ответ `E`*NN*	
установить скорость **(не рекомендовано)**	b*бод*	Установить скорость последовательной линии в *бод*.
установить точку останова **(не рекомендовано)**	B*адрес*,*режим*	Установить (*режим* ‘`S`’) или удалить (*режим* ‘`C`’) точку останова по адресу *адрес*. *Это было замещено пакетами ‘`Z`’ и ‘`z`’.*
продолжить	c*адрес*	*Адрес*—это адрес для возобновления выполнения. Если он опущен, возобновить с текущего адреса.
	ответ	смотрите ниже

продолжить с сигналом	C*сиг*;*адрес*	Продолжить с сигналом *сиг* (шестнадцатеричный номер сигнала). Если ‘;*адрес*’ опущено, выполнение возобновляется с прежнего адреса.
	ответ	смотрите ниже
переключить режим отладки **(не рекомендовано)**	d	переключить флаг отладки.
отсоединиться	D	Отсоединить GDB от удаленной системы. Посылается удаленной системе перед тем, как GDB отсоединится.
	ответ *нет ответа*	GDB не ждет никакого ответа после посылки этого пакета.
зарезервировано	e	Зарезервировано для использования в будущем
зарезервировано	E	Зарезервировано для использования в будущем
зарезервировано	f	Зарезервировано для использования в будущем
зарезервировано	F	Зарезервировано для использования в будущем
чтение регистров	g	Чтение регистров общего назначения.
	ответ *XX...*	Каждый байт данных регистра описывается двумя шестнадцатеричными цифрами. Они передаются с целевым порядком байтов. Размер каждого регистра и его позиция внутри *пакета* ‘g’ определяются внутренними макросами GDB *REGISTER_RAW_SIZE* и *REGISTER_NAME*. Спецификация нескольких стандартных пакетов ‘g’ приведена ниже.
	E*NN*	в случае ошибки.
запись в регистры	G*XX...*	Смотрите ‘g’ для описания данных *XX...* .
	ответ OK	в случае успеха
	ответ E*NN*	в случае ошибки
зарезервировано	h	Зарезервировано для использования в будущем

выбрать нить	H*ct...*	Установить нить для последующих операций (‘m’, ‘M’, ‘g’, ‘G’, и другие). *c* = ‘c’ для нитей, используемых при пошаговом выполнении и продолжении; *t...* может быть -1 для всех нитей. *c* = ‘g’ для нитей, используемых в других операциях. Если ноль—выбрать любую нить.
	ответ OK	в случае успеха
	ответ E*NN*	в случае ошибки
пошаговое выполнение по тактовому циклу **(черновик)**	i*адрес*,*nnn*	Выполнить один тактовый цикл на удаленной машине. Если ‘,*nnn*’ указано, выполнить *nnn* циклов. Если указан *адрес*, пошаговое выполнение по одному тактовому циклу начинается этого адреса.
сигнал, затем выполнение по тактовым циклам **(зарезервировано)**	I	Смотрите ‘i’ и ‘S’, там аналогичный синтаксис и семантика.
зарезервировано	j	Зарезервировано для использования в будущем
зарезервировано	J	Зарезервировано для использования в будущем
убить	k	FIXME: *Нет описания, как действовать в случае, если был выбран контекст определенной нити (то есть ‘k’ убивает только эту нить?).*
зарезервировано	l	Зарезервировано для использования в будущем
зарезервировано	L	Зарезервировано для использования в будущем
чтение памяти	m*адрес*,*длина*	Прочитать *длину* байт памяти, начиная с адреса *адрес*. Ни GDB, ни заглушка не предполагают, что передача области памяти происходит по адресам, выровненным по границе слова. FIXME: *Нужен механизм передачи области памяти, выровненной по границе слова.*

	ответ *XX...*	*XX...* представляет собой содержимое памяти. Может содержать меньше запрошенного числа байт, если удалось прочитать только часть данных. Ни GDB, ни заглушка не предполагают, что передача области памяти происходит по адресам, выровненным по границе слова. FIXME: *Нужен механизм передачи области памяти, выровненной по границе слова.*
	ответ E*NN*	*NN* представляет номер ошибки
запись в память	M*адрес*,*длина*:*XX...*	Записать *длину* байт памяти, начиная с адреса *адрес*. *XX...*—это данные.
	ответ OK	при успехе
	ответ E*NN*	при ошибке (это включает случай, когда была записана только часть данных).
зарезервировано	n	Зарезервировано для использования в будущем
зарезервировано	N	Зарезервировано для использования в будущем
зарезервировано	o	Зарезервировано для использования в будущем
зарезервировано	O	Зарезервировано для использования в будущем
чтение регистров **(зарезервировано)**	p*n...*	Смотрите запись регистров.
	возврат *r....*	Значение регистра в целевом порядке байт, закодированное в шестнадцатеричном виде.
запись регистров	P*n...*=*r...*	Записать в регистр *n...* значение *r...*, которое содержит две шестнадцатеричные цифры для каждого байта в регистре (целевой порядок байтов).
	ответ OK	в случае успеха
	ответ E*NN*	при ошибке

общий запрос	`q`*запрос*	Запросить информацию о *запросе*. Вообще, запросы GDB имеют первую заглавную букву. Специальные запросы от производителей должны использовать приставку компании (из маленьких букв). Например: '`qfsf.var`'. За *запросом* может следовать необязательный список, разделенный '`,`' или '`;`'. Заглушки должны проверять, что они производят сравнение с полным именем *запроса*.
	ответ `XX...`	Данные от запроса, закодированные шестнадцатеричными цифрами. Ответ не может быть пустым.
	ответ `E`*NN* ответ ''	ответ при ошибке Указывает на нераспознанный *запрос*.
общая установка	`Q`*перем*`=`*знач*	Установить значение *перем* в *знач*. Смотрите '`q`' для обсуждения соглашений, касающихся имен.
сброс **(не рекомендовано)**	`r`	Установка всей системы в исходное состояние.
удаленная перезагрузка	`R`*XX*	Перезапустить удаленный сервер. *XX*, где оно требуется, не имеет ясного определения. FIXME: *Нужен пример взаимодействия, объясняющий как эти пакеты используются в расширенном удаленном режиме.*
пошагавое выполнение	`s`*адрес*	*Адрес*—это адрес для возобновления выполнения. Если *адрес* опущен, возобновить выполнение с того же адреса.
	ответ	смотрите ниже
пошаговое выполнение с сигналом	`S`*сиг*`;`*адрес*	Как `C`, но разница такая же, как между `step` и `continue`.
	ответ	смотрите ниже
поиск	`t`*адрес*`:`*PP*`,`*MM*	Поиск в обратном направлении, начиная с адреса *адрес*, до совпадения с шаблоном *PP* и маской *MM*. *PP* и *MM*—4 байта. *Адрес* должен быть не менее трех цифр.
жива ли нить	`T`*XX*	Определить, жива ли нить *XX*.

	ответ `OK` ответ `E`*NN*	нить все еще жива нить мертва
зарезервировано	`u`	Зарезервировано для использования в будущем
зарезервировано	`U`	Зарезервировано для использования в будущем
зарезервировано	`v`	Зарезервировано для использования в будущем
зарезервировано	`V`	Зарезервировано для использования в будущем
зарезервировано	`w`	Зарезервировано для использования в будущем
зарезервировано	`W`	Зарезервировано для использования в будущем
зарезервировано	`x`	Зарезервировано для использования в будущем
запись в память (двоичная)	`X`*адрес*`,`*длина*`:`*XX...*	*Адрес* это адрес, *длина* это число байт, *XX...* это двоичные данные. Символы `$`, `#` и `0x7d` экранируются с помощью `0x7d`.
	ответ `OK` ответ `E`*NN*	в случае успеха в случае ошибки
зарезервировано	`y`	Зарезервировано для использования в будущем
зарезервировано	`Y`	Зарезервировано для использования в будущем
удалить точку останова или наблюдения **(черновик)**	`z`*t*`,`*адрес*`,`*длина*	Смотрите ‘`Z`’.

поместить точку останова или наблюдения **(черновик)**	`Z`*t*`,`*адрес*`,`*длина*	*t* представляет тип: ‘`0`’ в случае программной точки останова, ‘`1`’—аппаратная точка останова, ‘`2`’—точка наблюдения за записью, ‘`3`’—точка наблюдения за чтением, ‘`4`’—точка наблюдения за доступом; *адрес*—это адрес; *длина* задается в байтах. Для программной точки останова, *длина* задает размер инструкции, на которую надо поместить заплату. Для аппаратных точек останова и наблюдения, *длина* указывает размер области памяти для наблюдения. Чтобы избежать потенциальных проблем с повторными пакетами, операции должны быть идемпотентным образом.
	ответ `E`*NN*	в случае ошибки
	ответ `OK`	в случае успеха
	‘’	Если не поддерживается.
зарезервировано	<другое>	Зарезервировано для использования в будущем

Пакеты ‘`C`’, ‘`c`’, ‘`S`’, ‘`s`’ и ‘`?`’ могут получить в качестве ответа все нижеперечисленное. В случае пакетов ‘`C`’, ‘`c`’, ‘`S`’ и ‘`s`’, этот ответ возвращается только тогда, когда цель останавливается. Ниже, точное значение ‘**номера сигнала**’ определено нечетко. Вообще, используется одно из соглашений UNIX о номерах сигналов.

`S`*AA*	*AA*—это номер сигнала
`T`*AAn...*`:`*r...*`;`*n...*`:`*r...*`;`*n...*`:`*r...*`;`	*AA* = две шестнадцатеричные цифры номера сигнала; *n...* = (шестнадцатеричный) номер регистра, *r...* = содержимое регистра в целевом порядке байт, размер определяется `REGISTER_RAW_SIZE`; *n...* = ‘`thread`’, *r...* = идентификатор процесса нити, это шестнадцатеричное целое; *n...* = другая строка, не начинающаяся с шестнадцатеричной цифры. GDB должен игнорировать эту пару *n...*, *r...* и переходить к следующей. Таким образом мы можем расширять протокол.
`W`*AA*	Процесс завершается с кодом выхода *AA*. Это применимо только к определенным типам целей.
`X`*AA*	Процесс завершается с сигналом *AA*.

`N`*AA*;*t...*;*d...*;*b...* **(устарело)**	*AA* = номер сигнала; *t...* = адрес символа "_start"; *d...* = база раздела данных; *b...* = база раздела bss. *Примечание: используется только целями Cisco Systems. Разница между этим ответом и запросом "qOffsets" заключается в том, что пакет 'N' может прибыть самопроизвольно, тогда как запрос 'qOffsets' инициируется рабочим отладчиком.*
`O`*XX...*	*XX...*—шестнадцатеричное представление ASCII-данных. Это может произойти в любой момент, пока программа выполняется и отладчик должен продолжать ждать 'W', 'T', и т.п.

Следующие пакеты для установок и запросов уже были определены.

текущая нить	`qC`	Возвратить идентификатор текущей нити.
	ответ `QC`*идент-проц*	Где *идент-проц*—16-битный идентификатор процесса, представленный шестнадцатеричнами цифрами.
	ответ *	Любой другой ответ подразумевает старый идентификатор процесса.
идентификаторы всех нитей	`qfThreadInfo`	
	`qsThreadInfo`	Получить список идентификаторов активных нитей от целевой ОС. Так как число активных нитей может оказаться слишком большим и не поместиться в пакет ответа, этот запрос работает итерациями: он может требовать более одной последовательности запрос/ответ, для получения полного списка нитей. Первым запросом последовательности будет `qfThreadInfo`; последующими запросами последовательности будут запросы `qsThreadInfo`. Замечание: замещает запрос `qL` (смотрите ниже).
	ответ `m<`*ид*`>`	Идентификатор одной нити
	ответ `m<`*ид*`>,<`*ид*`>...`	список идентификаторов нитей, разделенных запятыми
	ответ `l`	(буква 'l' в нижнем регистре) обозначает конец списка. В ответ на каждый запрос, цель будет отвечать списком из разделенных запятыми идентификаторов нитей, в шестнадцатеричном представлении, с порядком байт от старшего. GDB будет отвечать на каждый ответ запросом других идентификаторов (используя форму `qs` запроса), пока цель не ответит `l` (буква 'l' в нижнем регистре, от английского слова `'last'`).
дополнительная информация о нити	`qThreadExtraInfo,`*ид*	

		Здесь *ид* является идентификатором нити в шестнадцатеричном представлении, в порядке байт от старшего. Получает печатаемое описание строки атрибутов нити от целевой ОС. Эта строка может содержать все что угодно, что целевая ОС сочтет интересным для GDB сообщить пользователю о нити. Эта строка выводится в отображении GDB '`info threads`'. Примерами возможной дополнительной информации являются "Runnable" или "Blocked on Mutex".
	ответ *XX...*	Где *XX...*—ASCII-данные в шестнадцатеричном представлении, содержащие печатную строку с дополнительной информацией об атрибутах нити.
запрос *список* или *список-нитей* **(не рекомендовано)**	`qL`*нач-флаг**число-нитей**след-нить*	
		Получить информацию о нити от операционной системы, где происходит выполнение. Здесь: *нач-флаг* (одна шестнадцатеричная цифра) есть единица, что указывает на первый запрос, или ноль, что определяет последующий запрос; *число-нитей* (две шестнадцатеричные цифры)—максимальное число нитей, которое может содержать пакет ответа; и *след-нить* (восемь шестнадцатеричных цифр), для последующих запросов (*нач-флаг* равен нулю), возвращается в ответ как *арг-нить*. Замечание: этот запрос был замещен запросом `qfThreadInfo` (смотрите выше).
	ответ `q``M`*число**конец**арг-нить**нить...*	
		Здесь: *число* (две шестнадцатеричные цифры)—число возвращаемых нитей; *конец* (одна шестнадцатеричная цифра), есть ноль, который определяет, что есть еще нити, и единица, определяющая, что больше нитей нет; *арг-нить* (восемь шестнадцатеричных цифр) представляет собой *след-нить* из пакета запроса; *нить...*—это последовательность идентификаторов нитей от цели. *Идент-нити* (восемь шестнадцатеричных цифр). Смотрите `remote.c:parse_threadlist_response()`.
вычислить CRC блока памяти	`qCRC:`*адрес*`,`*длина*	
	ответ `E`*NN*	Ошибка (например, ошибка доступа к памяти)
	ответ `C`*CRC32*	Лишняя 32-битная циклическая проверка указанной области памяти.
запросить смещения разделов	`qOffsets`	Получить смещения разделов, которые целевая машина использовала при повторном размещении загруженного образа. *Замечание: если смещение* `Bss` *включено в ответ, GDB это игнорирует и вместо этого применяет к разделу* `Bss` *смещение* `Data`.
	ответ `Text=`*xxx*`;Data=`*yyy*`;Bss=`*zzz*	

запросить информацию о нити	qP*режимидент-нити*	
		Возвращает информацию об *идент-нити*. Здесь: *режим* является 32-битным режимом в шестнадцатеричном представлении; *идент-нити*—64-битный идентификатор нити в шестнадцатеричном представлении.
	ответ *	Смотрите `remote.c:remote_unpack_thread_info_response()`.
удаленная команда	qRcmd,*КОМАНДА*	
		КОМАНДА (в шестнадцатеричном представлении) передается для выполнения локальному интерпретатору. Неверные команды должны сообщаться при помощи выходной строки. Перед конечным результирующим пакетом, целевая машина может также ответить некоторым количеством промежуточных O*ВЫВОД* пакетов вывода на консоль. *Разработчики должны учесть, что предоставление доступа к интерпретатору заглушки может иметь последствия для безопасности.*
	ответ OK	Ответ на команду без вывода.
	ответ *ВЫВОД*	Ответ на команду со строкой вывода *ВЫВОД*, в шестнадцатеричном представлении.
	ответ E*NN*	Указывает на неправильно сформированный запрос.
	reply ‘’	Когда ‘q’‘Rcmd’ не распознана.

Следующие пакеты ‘g’/‘G’ были определены раньше. Ниже, некоторые 32-битные регистры передаются в виде 64 бит. Эти регистры должны быть расширены нулем/знаком (как?), чтобы заполнять выделенное место. Байты регистра передаются в целевом порядке байтов. Две части в байте регистра передаются от более значимого к менее значимому.

MIPS32	Все регистры передаются как 32-битные величины в таком порядке: 32 общего назначения; sr; lo; hi; bad; cause; pc; 32 регистра с плавающей точкой; fsr; fir; fp.
MIPS64	Все регистры передаются как 64-битные величины (включая такие 32-битные регистры, как `sr`). Порядок такой же, как для `MIPS32`.

Вот пример последовательности для перезапускаемой цели. Заметьте, что перезапуск не получает никакого непосредственного вывода:

```
<- R00
-> +
target restarts
<- ?
-> +
-> T001:1234123412341234
<- +
```

Пример последовательности при при пошаговом выполнении цели по одной инструкции:

```
<- G1445...
```

```
-> +
<- s
-> +
time passes
-> T001:1234123412341234
<- +
<- g
-> +
-> 1455...
<- +
```

13.4.1.5 Использование программы gdbserver

`gdbserver` является управляющей программой для Unix-подобных систем, которая позволяет вам установить соединение вашей программы с удаленным GDB посредством `target remote`, но без компоновки с обычной отладочной заглушкой.

`gdbserver` не является полной заменой отладочных заглушек, потому что требует по существу тех же средств операционной системы, что и сам GDB. Фактически, система, на которой может выполняться `gdbserver` для соединения с удаленным GDB, может также выполнять GDB локально! Тем не менее, `gdbserver` иногда полезен, так как по размеру эта программа гораздо меньше, чем GDB. `gdbserver` также легче переносить, чем весь GDB, так что вы сможете быстрее начать работать в новой системе, используя его. Наконец, если вы разрабатываете программы для систем реального времени, вы можете обнаружить, что накладные расходы, связанные с операциями реального времени, делают более удобным проведение всей возможной разработки на другой системе, например, с помощью кросс-компиляции. Вы можете использовать `gdbserver`, чтобы реализовать аналогичный выбор для отладки.

GDB и `gdbserver` общаются или через последовательную линию, или через соединение TCP, используя стандартный удаленный последовательный протокол GDB.

На целевой машине

вам необходимо иметь копию программы, которую вы хотите отладить. `gdbserver` не нуждается в таблице символов вашей программы, так что вы можете ее исключить, если необходимо сохранить пространство. Всю обработку символов осуществляет GDB на рабочей машине.

Чтобы использовать сервер, вы должны сообщить ему, как взаимодействовать с GDB, имя вашей программы и ее аргументы. Синтаксис следующий:

```
target> gdbserver comm программа [ арг ... ]
```

comm—это или имя устройства (для использования последовательной линии), или имя рабочей машины и номер порта TCP. Например, для отладки Emacs с параметром '`foo.txt`' и взаимодействия с GDB через последовательный порт '`/dev/com1`':

```
target> gdbserver /dev/com1 emacs foo.txt
```

`gdbserver` пассивно ждет рабочего GDB для связи с ним.

При использовании TCP-соединения вместо последовательной линии:

```
target> gdbserver host:2345 emacs foo.txt
```

Единственное отличие от предыдущего примера состоит в первом параметре, определяющем, что вы связываетесь с рабочим GDB через TCP. Параметр '`host:2345`' означает, что `gdbserver` должен ожидать TCP-соединение от машины '`host`' к локальному порту TCP 2345. (В настоящее время часть '`host`' игнорируется.) Вы можете выбрать любой номер порта, какой захотите, если

при этом он не конфликтует с какими-либо портами TCP, уже использующимися на целевой системе (например, 23 зарезервирован для `telnet`).[1] Вы должны использовать тот же номер порта с командой рабочего GDB `target remote`.

На рабочей машине GDB

вам нужна копия вашей программы с символьными данными, так как GDB нужна информация о символах и отладочная информация. Запустите GDB как обычно, используя имя локальной копии вашей программы в качестве первого аргумента. (Вам также может понадобиться ключ '`-baud`', если последовательная линия работает на скорости, отличной от 9600 бит/сек.) После этого, используйте `target remote`, чтобы установить связь с `gdbserver`. Ее параметры—либо имя устройства (обычно последовательного устройства, такого как '`/dev/ttyb`'), либо дескриптор порта TCP в форме *машина*`:`*порт*. Например:

```
(gdb) target remote /dev/ttyb
```

взаимодействует с сервером через последовательную линию '`/dev/ttyb`', а

```
(gdb) target remote the-target:2345
```

взаимодействует через TCP-соединение с портом 2345 на рабочей машине '`the-target`'. Для TCP-соединения, вы должны запустить `gdbserver` до использования команды `target remote`. Иначе вы можете получить ошибку, текст которой зависит от рабочей системы, но обычно он выглядит примерно так: '`Connection refused`'.

13.4.1.6 Использование программы `gdbserve.nlm`

`gdbserve.nlm`—это управляющая программа для систем NetWare, которая позволяет вам установить соединение вашей программы с удаленным GDB посредством `target remote`.

GDB и `gdbserve.nlm` общаются через последовательную линию, используя стандартный удаленный последовательный протокол GDB.

На целевой машине

вам необходимо иметь копию программы, которую вы хотите отладить. `gdbserve.nlm` не нуждается в таблице символов вашей программы, так что вы можете ее исключить, если необходимо сохранить пространство. GDB осуществляет всю обработку символов на рабочей машине.

Чтобы использовать сервер, вы должны сообщить ему как взаимодействовать с GDB, имя вашей программы и ее аргументы. Синтаксис следующий:

```
load gdbserve [ BOARD=плата ] [ PORT=порт ]
              [ BAUD=бод ] программа [ арг ... ]
```

Плата и *порт* определяют последовательную линию; *бод* определяет скорость в бодах, используемую соединением. Значения *порт* и *node* по умолчанию равны 0, *бод* по умолчанию 9600 бит/сек.

Например, для отладки Emacs с параметром '`foo.txt`' и взаимодействия с GDB через последовательный порт номер 2 на плате 1, используя соединение 19200 бит/сек:

```
load gdbserve BOARD=1 PORT=2 BAUD=19200 emacs foo.txt
```

На рабочей машине GDB

вам нужна копия вашей программы с символьными данными, так как GDB требуется символьная и отладочная информация. Запустите GDB как обычно,

[1] Если вы выберете номер порта, который конфликтует с другим сервисом, `gdbserver` печатает сообщение об ошибке и завершает работу.

используя имя локальной копии вашей программы в качестве первого параметра. (Вам также может понадобиться ключ '`-baud`', если последовательная линия работает на скорости, отличной от 9600 бит/сек.) После этого, используйте `target remote` для установки связи с `gdbserve.nlm`. Ее аргумент—имя устройства (обычно последовательного устройства, такого как '`/dev/ttyb`'). Например:

```
(gdb) target remote /dev/ttyb
```

соединение с сервером через последовательную линию '`/dev/ttyb`'.

13.5 Отображение объектов ядра

Некоторые цели поддерживают отображение объектов ядра. При помощи этих возможностей, GDB взаимодействует непосредственно с операционной системой и может выводить информацию об объектах уровня операционной системы, например, о блокировках и других объектах синхронизации. Какие именно объекты могут быть отображены, определяется в зависимости от конкретной ОС.

Используйте команду `set os`, чтобы установить тип операционной системы. Это говорит GDB, какой модуль отображения объектов ядра инициализировать:

```
(gdb) set os cisco
```

Если команда `set os` выполнится успешно, GDB выведет некоторую информацию об операционной системе, и создаст новую команду `info`, которая может быть использована для посылки запросов на целевую машину. Название команды `info` выбирается в зависимости от операционной системы:

```
(gdb) info cisco
List of Cisco Kernel Objects
Object     Description
any        Any and all objects
```

Дальнейшие подкоманды могут использоваться для запросов о конкретных объектах, информация о которых есть в ядре.

В настоящее время не существует другого способа определить, поддерживается та или иная операционная система, кроме как попробовать.

14 Информация о конфигурации

В то время как почти все команды GDB доступны для всех чистых и кросс-версий отладчика, существуют некоторые исключения. Эта глава описывает вещи, доступные только в определенных конфигурациях.

Существует три основные категории конфигураций: чистые конфигурации, где рабочая и целевая машины совпадают, конфигурации для встроенных операционных систем, которые обычно совпадают для нескольких различных архитектур процессоров, и отдельные встроенные процессоры, которые сильно отличаются друг от друга.

14.1 Чистая конфигурация

Этот раздел описывает детали, специфичные для определенных чистых конфигураций.

14.1.1 HP-UX

В системах HP-UX, если вы ссылаетесь на функцию или переменную, имя которой начинается со знака доллара, GDB сперва ищет имя пользователя или системы, до поиска вспомогательной переменной.

14.1.2 Информация о процессах SVR4

Многие версии SVR4 предоставляют возможность, называемую '`/proc`', которая может быть использована для исследования образа выполняемого процесса, используя подпрограммы файловой системы. Если GDB сконфигурирован для операционной системы, поддерживающей эту возможность, команда `info proc` доступна для получения отчета по некоторым видам информации о процессе, выполняющем вашу программу. `info proc` работает только на системах SVR4, которые включают код `procfs`. Среди этих систем: OSF/1 (Digital Unix), Solaris, Irix и Unixware, но не HP-UX или Linux, к примеру.

`info proc` Выдает доступную суммарную информацию о процессе.

`info proc mappings`
: Сообщает диапазоны адресов, доступных в программе, с информацией, может ли ваша программа читать, записывать, или исполнять каждый из диапазонов.

`info proc times`
: Время запуска, время пользовательского и системного ЦП для вашей программы и ее потомков.

`info proc id`
: Сообщает информацию об идентификаторах процессов, относящихся к вашей программе: ее собственный идентификатор, идентификатор ее родителя, группы процесса и сеанса.

`info proc status`
: Общая информация о состоянии процесса. Если процесс остановлен, то этот отчет включает причину останова и любые полученные сигналы.

`info proc all`
: Показывает всю вышеперечисленную информацию о процессе.

14.2 Встроенные операционные системы

Этот раздел описывает конфигурации, задействующие отладку встроенных операционных систем, которые доступны для нескольких различных архитектур.

GDB включает возможность отлаживать программы, выполняющиеся в различных операционных системах реального времени.

14.2.1 Использование GDB с VxWorks

`target vxworks` *имя-машины*
: Система VxWorks, присоединенная посредством TCP/IP. Аргумент *имя-машины* есть имя или IP-адрес машины целевой системы.

На VxWorks, `load` компонует *имя-файла* динамически на текущей целевой системе, и добавляет ее символьную информацию в GDB.

GDB позволяет разработчикам запускать и отлаживать с Unix-машин задачи, выполняющиеся на сетевых целях VxWorks. Уже выполняющиеся задачи, запущенные из оболочки VxWorks, также могут быть отлажены. GDB использует код, который может выполняться как на машине Unix, так и на целевой машине VxWorks. Программа `gdb` устанавливается и выполняется на Unix-машине. (Она может быть установлена под именем `vxgdb`, чтобы отличать ее от GDB для отладки программ на рабочей машине.)

`VxWorks-timeout` *arг*
: Сейчас все цели, базирующиеся на VxWorks, поддерживают параметр `vxworks-timeout`. Этот параметр устанавливается пользователем, и *arг* представляют число секунд, в течение которых GDB ожидает ответы на вызовы удаленных процедур. Вы можете использовать это, если ваша целевая машина VxWorks является медленным программным эмулятором, или находится далеко на другом конце медленного сетевого соединения.

Следующая информация о соединении к VxWorks была свежей, когда это руководство было написано; более новые выпуски VxWorks могут использовать обновленные процедуры.

Для использования GDB с VxWorks, вы должны пересобрать ваше ядро VxWorks, чтобы включить подпрограммы интерфейса удаленной отладки в библиотеку VxWorks '`rdb.a`'. Чтобы это сделать, определите `INCLUDE_RDB` в конфигурационном файле VxWorks '`configAll.h`' и пересоберите ядро VxWorks. Получившееся ядро содержит '`rdb.a`', и порождает задачу отладки исходного кода `tRdbTask`, когда VxWorks загружается. Для большей информации по конфигурированию и сборке VxWorks, смотрите руководство изготовителя.

Когда вы включили '`rdb.a`' в образ вашей системы VxWorks и так установили ваши пути поиска выполняемых файлов, чтобы можно было найти GDB, вы готовы к вызову отладчика. Из вашей рабочей Unix-машины, выполните `gdb` (или `vxgdb`, в зависимости от вашей установки).

GDB появляется и показывает приглашение:

```
(vxgdb)
```

14.2.1.1 Соединение к VxWorks

Команда GDB `target` позволяет вам соединяться с целевой машиной VxWorks в сети. Для соединения с целью, имя которой есть "`tt`", введите:

```
(vxgdb) target vxworks tt
```

GDB покажет сообщения, аналогичные этим:

```
Attaching remote machine across net...
Connected to tt.
```

Затем GDB пытается считать таблицы символов всех объектных модулей, загруженных на целевой машине VxWorks с того момента, как она была включена. GDB находит эти файлы путем поиска в каталогах, перечисленных в путях поиска команд (см. Раздел 4.4 [Рабочая среда вашей программы], с. 23); если ему не удается найти объектный файл, он выводит подобное сообщение:

```
prog.o: No such file or directory.
```

Когда это происходит, добавьте соответствующий каталог к путям поиска с помощью команды GDB `path`, и выполните команду `target` снова.

14.2.1.2 Загрузка на VxWorks

Если вы соединились с целевой машиной VxWorks и хотите отладить объект, который еще не был загружен, вы можете использовать команду GDB `load`, чтобы загрузить файл из Unix в VxWorks. Объектный файл, заданный в качестве аргумента к `load`, в действительности открывается дважды: сначала целевой машиной VxWorks, чтобы загрузить код, а затем GDB, чтобы считать таблицу символов. Это может привести к проблемам, если текущие рабочие каталоги в этих системах различаются. Если обе системы монтируют по NFS одинаковые файловые системы, вы можете избежать этих проблем, используя абсолютные пути. В противном случае, проще всего установить рабочий каталог на обеих системах в тот, в котором расположен объектный файл, и затем ссылаться на него по имени, без пути. Например, программа '`prog.o`' может находиться в '*vxpath*`/vw/demo/rdb`' на VxWorks и в '*hostpath*`/vw/demo/rdb`' на рабочей машине. Для загрузки этой программы, введите в VxWorks следующее:

```
-> cd "vxpath/vw/demo/rdb"
```

Затем, в GDB, введите:

```
(vxgdb) cd hostpath/vw/demo/rdb
(vxgdb) load prog.o
```

GDB отобразит ответ, аналогичный этому:

```
Reading symbol data from wherever/vw/demo/rdb/prog.o... done.
```

Вы также можете использовать команду `load`, чтобы заново загрузить объектный модуль, после редактирования и повторной компиляции соответствующего исходного файла. Заметьте, что при этом GDB удаляет все определенные точки останова, автоматические отображения, вспомогательные переменные, и очищает историю значений. (Это необходимо для того, чтобы сохранить целостность структур данных отладчика, которые ссылаются на таблицу символов целевой системы.)

14.2.1.3 Запуск задач

Вы также можете присоединиться к существующей задаче, используя команду `attach` следующим образом:

```
(vxgdb) attach задача
```

где *задача* является шестнадцатеричным идентификатором задачи VxWorks. Когда вы присоединяетесь к задаче, она может выполняться либо быть приостановленной. Выполняющаяся задача приостанавливается в момент присоединения.

14.3 Встроенные процессоры

Этот раздел описывает детали, специфичные для определенных встроенных конфигураций.

14.3.1 Встроенный AMD A29K

`target adapt` *устр*
: Монитор Adapt для A29K.

`target amd-eb` *устр скорость прог*
: Удаленная PC-резидентная плата AMD EB29K, присоединенная по последовательным линиям. *Устр* является последовательным устройством, также как для `target remote`; *скорость* позволяет вам указать скорость линии; а *прог* является именем программы, которая будет отлаживаться, так, как оно появляется в ДОС на ПК. См. Раздел 14.3.1.2 [Протокол EBMON для AMD29K], с. 136.

14.3.1.1 A29K UDI

Для отладки процессоров семейства a29k, GDB поддерживает протокол AMD UDI ("Universal Debugger Interface"[1]). Для использования этой конфигурации с целями AMD, на которых выполняется монитор MiniMON, вам нужна программа `MONTIP`, доступная бесплатно у AMD. Вы можете также использовать GDB с программой `ISSTIP`, UDI-совместимым эмулятором a29k, также доступным у AMD.

`target udi` *кл-слово*
: Выбрать интерфейс UDI к удаленной плате a29k или эмулятору. Здесь *кл-слово* является элементом в конфигурационном файле AMD '`udi_soc`'. Этот файл содержит в качестве элементов ключевые слова, которые определяют параметры, используемые при соединении к целям a29k. Если файл '`udi_soc`' не находится в вашем рабочем каталоге, вы должны установить путь к нему в переменной среды '`UDICONF`'.

14.3.1.2 Протокол EBMON для AMD29K

AMD распространяет плату разработки 29K, предназначенную для помещения в ПК, вместе с программой монитора `EBMON`, работающей в ДОС. Коротко эта система разработки называется "EB29K". Чтобы использовать GDB из Unix-системы для выполнения программ на плате EB29K, вы должны сперва соединить последовательным кабелем ПК (в котором установлена плата EB29K) и последовательный порт на Unix-системе. Далее мы предполагаем, что вы соединили кабелем порт ПК '`COM1`' и '`/dev/ttya`' на Unix-системе.

14.3.1.3 Установка связи

Следующим шагом нужно установить параметры порта ПК, сделав в ДОС что-то вроде этого:

```
C:\> MODE com1:9600,n,8,1,none
```

Этот пример, выполненный в системе MS DOS 4.0, устанавливает порт ПК в 9600 бит/сек, без проверки четности, восьмибитные данные, один стоп-бит и отсутствие действия для "повтора"; вы должны использовать те же параметры связи при установке соединения со стороны Unix.

Чтобы передать управление с ПК стороне Unix, введите следующее в консоли ДОС:

[1] Универсальный отладочный интерфейс. *(Прим. переводчика)*

```
C:\> CTTY com1
```

(Позже, если вы хотите вернуть управление консоли ДОС, вы можете использовать команду `CTTY con`—но вы должны послать ее через устройство, имевшее управление, в нашем примере через последовательную линию '`COM1`'.)

На Unix-машине, для связи с ПК используйте коммуникационную программу, такую как `tip` или `cu`. Например

```
cu -s 9600 -l /dev/ttya
```

Показанные ключи для `cu` определяют, соответственно, скорость линии и последовательный порт. Если вместо этого вы используете `tip`, ваша командная строка может выглядеть следующим образом:

```
tip -9600 /dev/ttya
```

Ваша система может требовать другого имени в том месте, где мы показываем '`/dev/ttya`' в качестве аргумента к `tip`. Параметры связи, включая используемый порт, ассоциированы с аргументом к `tip` в файле описаний "remote"—обычно это '`/etc/remote`'.

Используя соединение `tip` или `cu`, измените рабочий каталог ДОС в тот, который содержит копию вашей программы 29K, затем запустите на ПК программу `EBMON` (управляющая программа EB29K, поставляемая AMD с вашей платой). Вы должны увидеть начальный вывод `EBMON`, аналогичный следующему, заканчивающийся приглашением `EBMON` '`#`':

```
C:\> G:

G:\> CD \usr\joe\work29k

G:\USR\JOE\WORK29K> EBMON
Am29000 PC Coprocessor Board Monitor, version 3.0-18
Copyright 1990 Advanced Micro Devices, Inc.
Written by Gibbons and Associates, Inc.

Enter '?' or 'H' for help

PC Coprocessor Type   = EB29K
I/O Base              = 0x208
Memory Base           = 0xd0000

Data Memory Size      = 2048KB
Available I-RAM Range = 0x8000 to 0x1fffff
Available D-RAM Range = 0x80002000 to 0x801fffff

PageSize              = 0x400
Register Stack Size   = 0x800
Memory Stack Size     = 0x1800

CPU PRL               = 0x3
Am29027 Available     = No
Byte Write Available  = Yes

# ~.
```

Затем выйдите из программы `cu` или `tip` (в этом примере это сделано при помощи ввода `~.` в приглашении `EBMON`). `EBMON` продолжает работать, готовый к тому, что GDB перехватит управление.

Для этого примера, мы предположили, что существует соединение PC/NFS, которое устанавливает файловую систему Unix-машины как “диск ‘G:’” на ПК. Это является, вероятно, самым удобным способом удостовериться, что одна и та же программа 29K находится и на ПК, и в Unix-системе. Если у вас нет PC/NFS или чего-нибудь аналогичного, соединяющего две системы, вы должны прибегнуть к другому способу передачи программы 29K из Unix на ПК—возможно переписать ее на дискету. GDB *не* загружает программы по последовательной линии.

14.3.1.4 Кросс-отладка EB29K

Наконец, перейдите в каталог, содержащий образ вашей программы 29K в Unix-системе, и запустите GDB, указав имя программы в качестве аргумента:

```
cd /usr/joe/work29k
gdb myfoo
```

Теперь вы можете использовать команду `target`:

```
target amd-eb /dev/ttya 9600 MYFOO
```

В этом примере мы предполагали, что ваша программа находится в файле ‘`myfoo`’. Заметьте, что имя файла, заданное в качестве последнего аргумента к `target amd-eb`, должно быть таким, каким его видит ДОС. В нашем примере, это просто `MYFOO`, но вообще оно может включать путь ДОС, и, в зависимости от механизма передачи, может быть не похоже на имя на Unix-машине.

В этом месте вы можете установить желаемые точки останова; когда вы будете готовы увидеть вашу программы выполняющейся на плате 29K, используйте команду GDB `run`.

Чтобы остановить отладку удаленной программы, используйте команду GDB `detach`.

Чтобы возвратить управление консоли ПК, используйте `tip` или `cu` снова, после завершения вашего сеанса GDB, чтобы присоединиться к `EBMON`. Затем вы можете ввести команду `q`, чтобы завершить работу `EBMON`, возвращая управление командному интерпретатору ДОС. Введите *`CTTY con`*, чтобы возвратить командный ввод основной консоли ДОС, и введите `~.`, чтобы покинуть `tip` или `cu`.

14.3.1.5 Удаленный журнал

Команда `target amd-eb` создает в текущем рабочем каталоге файл ‘`eb.log`’, чтобы помочь отладить проблемы с соединением. ‘`eb.log`’ записывает весь вывод ‘`EBMON`’, включая эхо посланных ему команд. Выполнение ‘`tail -f`’ для этого файла в другом окне часто помогает понять проблемы с `EBMON`, или неожиданные события на стороне ПК.

14.3.2 ARM

`target rdi` *устр*
: Монитор ARM Angel, через интерфейс библиотеки RDI к протоколу ADP. Вы можете использовать эту цель для взаимодействия как с платами, на которых выполняется монитор Angel, так и с устройством отладки EmbeddedICE JTAG.

`target rdp` *устр*
: Монитор ARM Demon.

14.3.3 Hitachi H8/300

`target hms` *устр*
: Плата Hitachi SH, H8/300 или H8/500, присоединенная через последовательную линию к вашей машине. Используйте специальные команды `device` и `speed` для управления последовательной линией и используемой скоростью связи.

`target e7000` *устр*
: Эмулятор E7000 для Hitachi H8 и SH.

`target sh3` *устр*
`target sh3e` *устр*
: Целевые системы Hitachi SH-3 и SH-3E.

Когда вы выбираете удаленную отладку для платы Hitachi SH, H8/300 или H8/500, команда `load` загружает вашу программу на плату Hitachi, и также открывает ее как текущую выполняемую цель для GDB на вашей машине (как команда `file`).

Для общения с вашим Hitachi SH, H8/300 или H8/500, GDB необходимо знать следующие вещи:

1. что вы хотите использовать: '`target hms`', удаленный отладочный интерфейс для микропроцессоров Hitachi, или '`target e7000`', встроенный эмулятор для Hitachi SH и Hitachi 300H. ('`target hms`' используется по умолчанию, если GDB сконфигурирован специально для Hitachi SH, H8/300 или H8/500.)
2. какое последовательное устройство соединяет вашу машину с платой Hitachi (по умолчанию используется первое последовательное устройство, доступное на вашей машине).
3. какую скорость использовать для этого последовательного устройства.

14.3.3.1 Соединение с платами Hitachi

Используйте специальную команду GDB '`device` *порт*', если вам нужно явно установить последовательное устройство. По умолчанию используется первый *порт*, доступный на вашей машине. Это необходимо только на Unix-машинах, где это обычно что-то типа '`/dev/ttya`'.

GDB имеет другую специальную команду для установки скорости связи: '`speed` *bps*'. Эта команда также используется только на Unix-машинах; в ДОС, устанавливайте скорость линии как обычно извне GDB командой `mode` (например, *`mode com2:9600,n,8,1,p`* для соединения 9600 бит/сек.

Команды '`device`' и '`speed`' доступны для отладки программ микропроцессора Hitachi, только если вы используете рабочую среду Unix. Если вы используете ДОС, для взаимодействия с платой разработки через последовательный порт ПК GDB полагается на вспомогательную резидентную программу `asynctsr`. Вы также должны использовать команду ДОС `mode`, чтобы подготовить порт со стороны ДОС.

Следующий пример сеанса иллюстрирует шаги, необходимые для запуска программы на H8/300 под управлением GDB. В нем используется программа H8/300 под названием '`t.x`'. Для Hitachi SH и H8/500 процедура та же самая.

Сперва подсоедините вашу плату разработки. В этом примере, мы используем плату, присоединенную к порту `COM2`. Если вы используете другой последовательный порт, подставьте его имя в агрументе команды `mode`. Когда вы вызываете `asynctsr`, вспомогательную программу связи, используемую отладчиком, вы передаете ей только числовую часть имени последовательного порта; например, ниже '`asynctsr 2`' запускает `asynctsr` для `COM2`.

```
C:\H8300\TEST> asynctsr 2
C:\H8300\TEST> mode com2:9600,n,8,1,p
```

```
Resident portion of MODE loaded

COM2: 9600, n, 8, 1, p
```

Предупреждение: Мы обнаружили ошибку в PC-NFS, которая конфликтует с `asynctsr`. Если вы также используете PC-NFS на вашей ДОС-машине, вам может потребоваться отключить его, или даже загрузить машину без него, чтобы использовать `asynctsr` для управления отладочной платой.

Теперь, когда связь установлена и плата разработки присоединена, вы можете запустить GDB. Вызовите `gdb` с именем вашей программы в качестве аргумента. GDB выводит обычное приглашение: '`(gdb)`'. Используйте две специальные команды для начала сеанса отладки: '`target hms`' для задания кросс-отладки для платы Hitachi, и команду `load` для загрузки вашей программы на нее. `load` выводит имена разделов программы, и '`*`' для каждых двух килобайт загруженных данных. (Если вы хотите обновить данные GDB для символов или для выполняемого файла без загрузки, используйте команды GDB `file` или `symbol-file`. Для описания этих команд, равно как и самой команды `load`, см. Раздел 12.1 [Команды для задания файлов], с. 105.)

```
(eg-C:\H8300\TEST) gdb t.x
GDB is free software and you are welcome to distribute copies
 of it under certain conditions; type "show copying" to see
 the conditions.
There is absolutely no warranty for GDB; type "show warranty"
for details.
GDB 5.0, Copyright 1992 Free Software Foundation, Inc...
(gdb) target hms
Connected to remote H8/300 HMS system.
(gdb) load t.x
.text   : 0x8000 .. 0xabde ***********
.data   : 0xabde .. 0xad30 *
.stack  : 0xf000 .. 0xf014 *
```

Теперь вы готовы выполнять или отлаживать вашу программу. С этого момента, вы можете использовать все обычные команды GDB. Команда `break` устанавливает точки останова; `run` запускает вашу программу; `print` или `x` отображает данные; команда `continue` возобновляет выполнение после остановки в точке останова. Вы можете использовать команду `help` в любой момент, чтобы узнать больше о командах GDB.

Помните, однако, что возможности *операционной системы* не доступны на вашей плате разработки; например, если ваша программа зависает, вы не можете послать сигнал прерывания—но можете нажать кнопку RESET!

Используйте кнопку RESET на вашей плате разработки

- чтобы прервать вашу программу (не используйте *`ctl-C`* на машине с ДОС—у нее нет способа передать сигнал прерывания на плату разработки); и
- для возврата к приглашению GDB после того, как ваша программа нормально завершается. Протокол связи не предусматривает другого способа для GDB определить, что ваша программа завершилась.

В любом случае, GDB видит результат нажатия RESET на плате разработки как "нормальное завершение" вашей программы.

14.3.3.2 Использование встроенного эмулятора E7000

Вы можете использовать встроенный эмулятор E7000 для разработки кода либо для Hitachi SH, либо для H8/300H. Используйте одну из этих форм команды ‘`target e7000`’ для соединения GDB с вашей E7000:

`target e7000` *порт скорость*
: Используйте эту форму, если ваша E7000 присоединена к последовательному порту. Аргумент *порт* идентифицирует, какой последовательный порт использовать (например, ‘`com2`’). Третий аргумент является скоростью линии в битах в секунду (например, ‘`9600`’).

`target e7000` *имя-узла*
: Если ваша E7000 установлена как узел сети TCP/IP, вы можете просто указать его имя; GDB использует для соединения `telnet`.

14.3.3.3 Специальные команды GDB для Hitachi

Некоторые команды GDB доступны только для H8/300:

`set machine h8300`
`set machine h8300h`
: Настраивайте GDB на один из двух вариантов архитектур H8/300 с помощью ‘`set machine`’. Вы можете использовать ‘`show machine`’, чтобы проверить, какой из вариантов действует в данный момент.

14.3.4 H8/500

`set memory` *мод*
`show memory`
: Укажите, какую модель памяти H8/500 (*мод*) вы используете с помощью ‘`set memory`’; проверяйте, какая модель используется при помощи ‘`show memory`’. Допустимыми значениями для *мод* являются `small`, `big`, `medium` и `compact`.

14.3.5 Intel i960

`target mon960` *устр*
: Монитор MON960 для Intel i960.

`target nindy` *имя-устр*
: Плата Intel 960, управляемая Nindy Monitor. *Имя-устр* является именем последовательного устройства, которое должно использоваться для соединения, например ‘`/dev/ttya`’.

Nindy—это программа ROM Monitor для целевых систем Intel 960. Когда GDB сконфигурирован для управления удаленным Intel 960 с использованием Nindy, вы можете указать ему, как присоединиться к 960 несколькими способами:

- Указав последовательный порт, версию протокола Nindy и скорость связи через ключи командной строки;
- Ответив на запрос при старте;
- Используя команду `target` в любом месте вашего сеанса GDB. См. Раздел 13.2 [Команды для управления целями], с. 111.

С интерфейсом Nindy к плате Intel 960, команда `load` загружает *имя-файла* на 960, а также добавляет его символьные данные в GDB.

14.3.5.1 Вызов Nindy

Если вы просто запустите `gdb` без использования ключей командной строки, у вас запросят, какой последовательный порт использовать, *до* того, как вы получите обычное приглашение GDB:

```
Attach /dev/ttyNN - specify NN, or "quit" to quit:
```

Ответьте на запрос с любым суффиксом (после '`/dev/tty`'), определяющим последовательный порт, который вы хотите использовать. Вы можете, по своему выбору, просто начать работу без соединения с Nindy, ответив на приглашение пустой строкой. Если вы сделаете это и позже захотите присоединиться к Nindy, используйте `target` (см. Раздел 13.2 [Команды для управления целями], с. 111).

14.3.5.2 Параметры для Nindy

Вот параметры вызова для начала вашего сеанса GDB с подключенной платой Nindy-960:

`-r` *порт*
: Задайте имя порта последовательного интерфейса, который должен использоваться для соединения с целевой системой. Этот ключ доступен только когда GDB сконфигурирован для целевой архитектуры Intel 960. Вы можете определить *порт* любым из следующих способов: полный путь (например, '`-r /dev/ttya`'), имя устройства в '`/dev`' (например, '`-r ttya`') или просто уникальный суффикс для определенного `tty` (например, '`-r a`').

`-O`
: (Заглавная буква "O", не ноль.) Определяет, что GDB должен использовать "старый" протокол монитора Nindy для соединения с целевой системой. Этот ключ доступен только когда GDB сконфигурирован для целевой архитектуры Intel 960.

  Предупреждение: если вы определите '`-O`', но в действительности попытаетесь связаться с системой, которая ожидает более нового протокола, соединение не будет установлено, как будто не соответствуют скорости. GDB неоднократно пытается соединиться снова на нескольких различных скоростях линии. Вы можете остановить этот процесс посредством прерывания.

`-brk`
: Определяет, что GDB должен сперва послать целевой системе сигнал `BREAK`, пытаясь сбросить ее, перед соединением с целью Nindy.

  Предупреждение: Многие целевые системы не имеют требуемых для этого аппаратных средств; это работает только на немногих платах.

Стандартный ключ '`-b`' управляет скоростью линии, используемой на последовательном порту.

14.3.5.3 Команда сброса Nindy

`reset`
: Для целей Nindy, эта команда посылает "break" удаленной целевой системе; она полезна, только если целевая система была оборудована схемой для выполнения аппаратного сброса (или других действий, представляющих интерес) при обнаружении прерывания.

14.3.6 Mitsubishi M32R/D

`target m32r` *устр*
: Монитор ROM Mitsubishi M32R/D.

14.3.7 M68k

Конфигурация Motorola m68k включает поддержку ColdFire, и команду `target` для следующих мониторов ROM.

`target abug` *устр*
: Монитор ABug ROM для M68K.

`target cpu32bug` *устр*
: Монитор CPU32BUG, выполняющийся на плате CPU32 (M68K).

`target dbug` *устр*
: Монитор dBUG ROM для Motorola ColdFire.

`target est` *устр*
: Монитор EST-300 ICE, выполняющийся на плате CPU32 (M68K).

`target rom68k` *устр*
: Монитор ROM 68K, выполняющийся на плате M68K IDP.

Если GDB сконфигурирован с `m68*-ericsson-*`, то вместо этого у него будет только одна специальная команда `target`:

`target es1800` *устр*
: Эмулятор ES-1800 для M68K.

`target rombug` *устр*
: Монитор ROMBUG ROM для OS/9000.

14.3.8 M88K

`target bug` *устр*
: Монитор BUG, выполняющийся на плате MVME187 (m88k).

14.3.9 Встроенный MIPS

GDB может использовать удаленный отладочный протокол MIPS для взаимодействия с платой MIPS, присоединенной к последовательной линии. Эта возможность доступна, если вы сконфигурировали GDB с '`-target=mips-idt-ecoff`'.

Используйте эти команды GDB для определения соединения с вашей целевой платой:

`target mips` *порт*
: Для выполнения программы на плате, выполните `gdb`, задав имя программы в качестве аргумента. Для соединения с платой, используйте команду '`target mips` *порт*', где *порт*—имя последовательного порта, присоединенного к плате. Если программа еще не была загружена на плату, вы можете использовать команду `load`, чтобы это сделать. Затем вы можете использовать все обычные команды GDB.

  Например, эта последовательность команд устанавливает соединение к целевой плате через последовательный порт, загружает и начинает выполнение из отладчика программы с именем *prog*:

```
host$ gdb prog
GDB is free software and ...
(gdb) target mips /dev/ttyb
(gdb) load prog
(gdb) run
```

`target mips` *имя-машины*`:`*номер-порта*
: В некоторых рабочих конфигурациях GDB, вы можете задать TCP-соединение (например, к последовательной линии, управляемой терминальным концентратором) вместо последовательного порта, используя синтаксис '*имя-машины*`:`*номер-порта*'.

`target pmon` *порт*
: Монитор ROM PMON.

`target ddb` *порт*
: NEC DDB-разновидность PMON для Vr4300.

`target lsi` *порт*
: LSI-разновидность PMON.

`target r3900` *устр*
: Densan DVE-R3900 монитор ROM для Toshiba R3900 Mips.

`target array` *устр*
: Плата контроллера RAID Array Tech LSI33K.

GDB также поддерживает следующие специальные команды для целей MIPS:

`set processor` *арг*
`show processor`
: Используйте команду `set processor` для установки типа процессора MIPS, когда вы хотите обратиться к регистрам, уникальным для данного типа процессора. Например, `set processor` *r3041* велит GDB использовать регистры CPO, соответствующие микросхеме 3041. Используйте команду `show processor`, чтобы узнать, какой процессор MIPS используется GDB. Используйте команду `info reg` чтобы узнать, какие регистры использует GDB.

`set mipsfpu double`
`set mipsfpu single`
`set mipsfpu none`
`show mipsfpu`
: Если ваша целевая плата не поддерживает сопроцессор MIPS для вычислений с плавающей точкой, вы должны использовать команду '`set mipsfpu none`' (если вам это нужно, вы можете поместить эту команду в ваш файл инициализации GDB). Это говорит GDB, как найти значения функций, которые возвращают величины с плавающей точкой. Это также позволяет GDB избежать сохранения регистров с плавающей точкой при вызове функций на плате. Если вы используете сопроцессор поддержки вычислений с плавающей точкой с поддержкой только одинарной точности, как на процессоре R4650, используйте команду '`set mipsfpu single`'. По умолчанию используется сопроцессор поддержки вычислений с плавающей точкой двойной точности; этот режим может быть выбран с помощью '`set mipsfpu double`'.

  В предыдущих версиях, единственным выбором была двойная точность или отсутствие поддержки вычислений с плавающей точкой, так что '`set mipsfpu on`' выберет режим двойной точности, а '`set mipsfpu off`' отключит эту поддержку.

  Как обычно, вы можете запросить значение переменной `mipsfpu` при помощи '`show mipsfpu`'.

`set remotedebug` *n*
`show remotedebug`
: Вы можете увидеть некоторую отладочную информацию о связи с платой, установив переменную `remotedebug`. Если вы установите ее в `1` при помощи '`set`

`remotedebug 1`', будет отображаться каждый пакет. Если вы установите ее в `2`, то будет отображаться каждый символ. В любой момент вы можете проверить текущее значение переменной командой '`show remotedebug`'.

`set timeout` *секунды*
`set retransmit-timeout` *секунды*
`show timeout`
`show retransmit-timeout`
: Вы можете управлять временем ожидания пакета, используемом в удаленном протоколе MIPS, при помощи команды `set timeout` *секунды*. Значение по умолчанию—5 секунд. Аналогично, вы можете управлять временем ожидания, используемом при ожидании подтверждения пакета с помощью команды `set retransmit-timeout` *секунды*. По умолчанию 3 секунды. Вы можете узнать обе эти величины с помощью `show timeout` и `show retransmit-timeout`. (Эти команды доступны *только* если GDB сконфигурирован для цели '`-target=mips-idt-ecoff`'.)

  Время ожидания, установленное при помощи `set timeout`, не имеет значения, когда GDB ожидает остановки вашей программы. В этом случае, GDB ждет бесконечно, потому что у него нет способа узнать, сколько программа будет выполняться, пока не остановится.

14.3.10 PowerPC

`target dink32` *устр*
: Монитор ROM DINK32.

`target ppcbug` *устр*
`target ppcbug1` *устр*
: Монитор ROM PPCBUG для PowerPC.

`target sds` *устр*
: Монитор SDS, выполняющийся на плате PowerPC (такой как Motorola ADS).

14.3.11 Встроенный HP PA

`target op50n` *устр*
: Монитор OP50N, выполняющийся на плате OKI HPPA.

`target w89k` *устр*
: Монитор W89K, выполняющийся на плате Winbond HPPA.

14.3.12 Hitachi SH

`target hms` *устр*
: Плата Hitachi SH, присоединенная через последовательную линию к вашей рабочей машине. Используйте специальные команды `device` и `speed` для управления последовательной линией и используемой скоростью связи.

`target e7000` *устр*
: Эмулятор E7000 для Hitachi SH.

`target sh3` *устр*
`target sh3e` *устр*
: Целевые системы Hitachi SH-3 и SH-3E.

14.3.13 Tsqware Sparclet

GDB позволяет разработчикам отлаживать с Unix-машины задачи, выполняющиеся на целевых системах Sparclet. GDB использует код, который выполняется как Unix-машине, так и на цели Sparclet. Программа `gdb` устанавливается и работает на Unix-машине.

`remotetimeout` *арг*
: GDB поддерживает параметр `remotetimeout`. Он устанавливатся пользователем, а *арг* представляет число секунд, в течение которых GDB ожидает ответы.

При компиляции для отладки, используйте ключи '`-g`' для получения отладочной информации, и '`-Ttext`' для того, чтобы разместить программу в том месте, в каком вы хотите загрузить ее на целевую машину. Вы также можете добавить ключ '`-n`' или '`-N`', чтобы уменьшить размеры разделов. Например:

```
sparclet-aout-gcc prog.c -Ttext 0x12010000 -g -o prog -N
```

Для проверки, что адреса в действительности являются теми, которые вы подразумевали, можно использовать `objdump`:

```
sparclet-aout-objdump -headers -syms prog
```

После того, как вы установили путь поиска выполняемых файлов, в котором присутствует GDB, вы готовы запустить отладчик. С вашей рабочей машины Unix, выполните `gdb` (или `sparclet-aout-gdb`, в зависимости от вашей установки).

GDB запустится и покажет приглашение:

```
(gdbslet)
```

14.3.13.1 Установка файла для отладки

Команда GDB `file` позволяет вам выбрать программу для отладки.

```
(gdbslet) file prog
```

Затем GDB пытается прочитать таблицу символов программы '`prog`'. Он находит файл путем поиска в каталогах, перечисленных в пути поиска команд. Если файл был скомпилирован с отладочной информацией (ключ '`-g`'), то также будет произведен поиск исходных файлов. GDB находит исходные файлы, производя поиск в каталогах, перечисленных в пути поиска каталогов (см. Раздел 4.4 [Рабочая среда вашей программы], с. 23). Если ему не удается найти файл, он выводит сообщение, подобное этому:

```
prog: No such file or directory.
```

Когда это случается, добавьте соответствующие каталоги в пути поиска с помощью команд GDB `path` и `dir`, и выполните команду `target` снова.

14.3.13.2 Соединение к Sparclet

Команда GDB `target` позволяет вам установить соединение с целевой машиной Sparclet. Для соединения с последовательным портом "`ttya`", введите:

```
(gdbslet) target sparclet /dev/ttya
Remote target sparclet connected to /dev/ttya
main () at ../prog.c:3
```

GDB выведет сообщение, подобное этому:

```
Connected to ttya.
```

14.3.13.3 Загрузка на Sparclet

Когда вы установили соединение к цели Sparclet, вы можете использовать команду GDB `load` для загрузки файла с рабочей машины на целевую. Имя файла и смещение загрузки должно быть задано команде `load` в качестве аргумента. Так как формат файла aout, программа должна быть загружена по начальному адресу. Чтобы определить, чему равна эта величина, вы можете использовать `objdump`. Смещение загрузки—это смещение, которое добавляется к VMA (Virtual Memory Address[2]) каждого раздела файла. Например, если программа '`prog`' была скомпонована с адресом текста 0x1201000, сегментом данных по адресу 0x12010160 и сегментом стека по адресу 0x12010170, введите в GDB:

```
(gdbslet) load prog 0x12010000
Loading section .text, size 0xdb0 vma 0x12010000
```

Если код загружается по адресу, отличному от того, по которому программа была скомпонована, вам может потребоваться использовать команды `section` и `add-symbol-file`, чтобы сообщить GDB, куда отобразить таблицу символов.

14.3.13.4 Выполнение и отладка

Теперь вы можете начать отлаживать задачу, используя команды GDB для управления выполнением, `b`, `step`, `run`, и так далее. Все такие команды перечислены в этом руководстве.

```
(gdbslet) b main
Breakpoint 1 at 0x12010000: file prog.c, line 3.
(gdbslet) run
Starting program: prog
Breakpoint 1, main (argc=1, argv=0xeffff21c) at prog.c:3
3        char *symarg = 0;
(gdbslet) step
4        char *execarg = "hello!";
(gdbslet)
```

14.3.14 Fujitsu Sparclite

`target sparclite` *устр*

Платы Fujitsu sparclite, используемые только с целью загрузки. Чтобы отлаживать программу, вы должны использовать дополнительную команду. Например, target remote *устр*, используя стандартный удаленный протокол GDB.

14.3.15 Tandem ST2000

GDB может быть использован с телефонным коммутатором Tandem ST2000, поддерживающим протокол Tandem STDBUG.

Для соединения вашего ST2000 с рабочей машиной, смотрите руководство производителя. После того, как ST2000 физически подключен, вы можете выполнить:

`target st2000` *устр скорость*

чтобы установить его как вашу отладочную среду. *Устр*—это обычно имя последовательного устройства, такое как '`/dev/ttya`', соединенного с ST2000 через последовательную линию. Вместо этого, вы можете указать *устр* как TCP-соединение (например, к последовательной линии, присоединенной через терминальный концентратор), используя синтаксис *имя-машины*:*номер-порта*.

[2] виртуальный адрес памяти *(Прим. переводчика)*

Команды `load` и `attach` *не* определены для этой цели; вы должны загрузить вашу программу на ST2000 также, как вы это обычно делаете для автономных действий. GDB читает отладочную информацию (например, символы) из отдельной, отладочной версии программы, которая доступна на вашем рабочем компьютере.

Следующие вспомогательные команды GDB доступны для облегчения работы в среде ST2000:

`st2000` *команда*
: Послать *команду* монитору STDBUG. Доступные команды описаны в руководстве производителя.

`connect`
: Соединяет управляющий терминал с командным монитором STDBUG. Когда вы закончили взаимодействие с STDBUG, ввод одной из двух последовательностей символов возвратит вас назад к приглашению GDB: ⟨RET⟩`~.` (Return, за которым следует тильда и точка) или ⟨RET⟩`~`⟨C-d⟩ (Return, за которым следует тильда и control-D).

14.3.16 Zilog Z8000

Будучи сконфигурированным для отладки целей Zilog Z8000, GDB включает имитатор Z8000.

Для семейства Z8000, '`target sim`' имитирует либо Z8002 (не сегментированный вариант архитектуры Z8000), либо Z8001 (сегментированный вариант). Имитатор распознает подходящую архитектуру изучая объектный код.

`target sim` *арг*
: Отладка программ на имитируемом ЦП. Если имитатор поддерживает параметры установки, укажите их в *арг*.

После определения этой цели, вы можете отлаживать программы для имитированного ЦП таким же образом, как программы для вашего рабочего компьютера; используйте команду `file` для загрузки образа новой программы, команду `run` для запуска вашей программы, и так далее.

Помимо того, что доступны все обычные машинные регистры (см. Раздел 8.10 [Регистры], с. 74), имитатор Z8000 предоставляет три специально названных регистра с дополнительной информацией:

`cycles` Считает тактовые импульсы с имитаторе.

`insts` Считает инструкции, выполненные в имитаторе.

`time` Время выполнения в шестидесятых долях секунды.

Вы можете ссылаться на эти значения в выражениях GDB с помощью обычных соглашений; например, '`b fputc if $cycles>5000`' устанавливает условную точку останова, которая срабатывает только после как минимум 5000 имитированных тактовых импульсов.

14.4 Архитектуры

Этот раздел описывает свойства архитектур, которые воздействуют на все применения GDB с данной архитектурой, как при чистой отладке, так и при кросс-отладке.

14.4.1 A29K

`set rstack_high_address` *адрес*
: В процессорах семейства AMD 29000, регистры сохраняются в отдельном *стеке регистров*. Для отладчика не существует способа определить размер этого стека. Обычно, GDB просто подразумевает, что стек “достаточно большой”. Это может привести к тому, что GDB попытается обратиться несуществующей области памяти. В случае необходимости, вы можете решить эту проблему, указав конечный адрес стека регистров с помощью команды `set rstack_high_address`. Аргумент должен быть адресом, который вы, вероятно, захотите начать с ‘`0x`’, чтобы задать его в шестнадцатеричном виде.

`show rstack_high_address`
: Отобразить текущее ограничение на стек регистров для процессоров семейства AMD 29000.

14.4.2 Alpha

Смотрите следующий раздел.

14.4.3 MIPS

Компьютеры, базирующиеся на архитектурах Alpha и MIPS, используют необычный кадр стека, который иногда требует от GDB поиска в объектном коде в обратном направлении, чтобы найти начало функции.

Чтобы сократить время ответа (особенно для встроенных приложений, где GDB может быть ограничен медленной последовательной линией для этого поиска), вы можете захотеть ограничить область поиска, используя одну из этих команд:

`set heuristic-fence-post` *предел*
: Ограничить GDB для исследования не более *предела* байт при поиске начала функции. Значение *0* (по умолчанию) означает неограниченный поиск. Однако, исключая *0*, чем больше предел, тем больше байт `heuristic-fence-post` должен просмотреть, и, следовательно, тем дольше он будет выполняться.

`show heuristic-fence-post`
: Отобразить текущее значение данного предела.

Эти команды доступны *только* когда GDB сконфигурирован для отладки программ на процессорах Alpha или MIPS.

15 Управление GDB

Вы можете изменять способы взаимодействия GDB с вами, используя команду `set`. Для команд, управляющих способами отображения данных GDB, смотрите Раздел 8.7 [Параметры вывода], с. 68. Другие установки описаны здесь.

15.1 Приглашение

GDB демонстрирует свою готовность считать команду, выводя строку, называемую *приглашением*. Обычно это '`(gdb)`'. Вы можете изменить строку приглашения командой `set prompt`. Например, при отладке GDB с помощью GDB, полезно изменить приглашение в одном из сеансов так, чтобы вы всегда могли понять, с каким из них вы общаетесь.

Замечание: `set prompt` не добавляет пробелы после установленного вами приглашения. Это позволяет устанавливать приглашение, заканчивающееся пробелом или нет.

`set prompt` *новое-пригл*
: Указывает GDB, что с этого момента надо использовать в качестве строки приглашения *новое-пригл*.

`show prompt`
: Печатает строку в форме: '`Gdb's prompt is:` *ваше-пригл*'

15.2 Редактирование команд

GDB читает входные команды через интерфейс *readline*. Эта библиотека GNU обеспечивает согласованное поведение для программ, которые предоставляют пользователю интерфейс командной строки. Преимуществами являются стили редактирования командной строки GNU Emacs или *vi*, `csh`-подобная подстановка истории и сохранение и повторное использование истории команд между сеансами отладки.

Вы можете управлять поведением редактирования командной строки в GDB командой `set`.

`set editing`
`set editing on`
: Включает редактирование командной строки (включено по умолчанию).

`set editing off`
: Отключает редактирование командной строки.

`show editing`
: Показывает, включено редактирование командной строки или нет.

15.3 История команд

GDB может отслеживать команды, которые вы вводите во время сеансов отладки, чтобы вы могли точно знать, что происходило. Используйте следующие команды для управления возможностями истории команд GDB.

`set history filename` *имя-файла*
: Устанавливает имя файла истории команд GDB в *имя-файла*. Это файл, откуда GDB читает исходный список истории команд и куда он записывает историю команд при выходе из данного сеанса. Вы можете обращаться к этому списку через раскрывание истории или с помощью символов редактирования

истории команд, перечисленных ниже. По умолчанию, этот файл есть значение переменной среды `GDBHISTFILE`, или, если эта переменная не установлена, '`./.gdb_history`' ('`./_gdb_history`' в MS-DOS).

`set history save`
`set history save on`
Записать историю команд в файл, имя которого может быть определено командой `set history filename`. По умолчанию, эта возможность отключена.

`set history save off`
Прекратить запись истории команд в файл.

`set history size` *размер*
Установить число команд, которые GDB хранит в своем списке истории. Значение по умолчанию берется из переменной среды `HISTSIZE`, или приравнивается 256, если эта переменная не установлена.

Раскрывание истории назначает специальное значение знаку `!`.

Так как `!` является также оператором логического отрицания в Си, раскрывание истории по умолчанию отключено. Если вы решите включить раскрывание истории командой `set history expansion on`, вы должны будете снабдить `!` (когда он используется как логическое отрицание в выражении) последующим пробелом или символом табуляции, чтобы предохранить его от раскрывания. Средства истории Readline не делают попытки подстановки на строках `!=` и `!(`, даже когда раскрывание истории включено.

Вот команды управления раскрыванием истории:

`set history expansion on`
`set history expansion`
Включить раскрывание истории. Раскрывание по умолчанию отключено.

`set history expansion off`
Отключить раскрывание истории.

Библиотека Readline поставляется вместе с более полной документацией по возможностям редактирования и раскрывания истории. Пользователи, незнакомые с GNU Emacs или `vi`, могут почитать ее.

`show history`
`show history filename`
`show history save`
`show history size`
`show history expansion`
Эти команды отображают состояние параметров истории GDB. Просто `show history` показывает все четыре состояния.

`show commands`
Отобразить последние десять команд в истории.

`show commands` *n*
Вывести десять команд, расположенных вокруг команды с номером *n*.

`show commands +`
Вывести десять команд, расположенных сразу после последних выведенных.

15.4 Размер экрана

Определенные команды GDB могут выводить большое количество информации на экран. Чтобы помочь вам всю ее прочитать, GDB приостанавливает вывод и запрашивает ввод в конце каждой страницы вывода. Нажмите ⟨RET⟩, когда вы хотите продолжить вывод, или `q` для уничтожения оставшегося вывода. Также, установка ширины экрана определяет, когда переносить строки вывода. В зависимости от того, что выводится, GDB пытается разбить строку в удобочитаемом месте, вместо того чтобы просто продолжить ее на следующую строчку.

Обычно GDB узнает размер экрана из программы драйвера терминала. Например, в Unix GDB использует базу данных termcap, вместе со значением переменной среды `TERM` и установками `stty rows` и `stty cols`. Если это неправильно, вы можете заменить эти установки командами `set height` и `set width`:

`set height` *lpp*
`show height`
`set width` *cpl*
`show width`

Эти команды `set` определяют высоту экрана в *lpp* строк и ширину в *cpl* знаков. Соответствующие команды `show` отображают текущие установки.

Если вы определите высоту в ноль строк, GDB не будет останавливаться при выводе, независимо от того, насколько он длинный. Это полезно, если вывод осуществляется в файл или буфер редактора.

Аналогично, вы можете определить '`set width 0`', чтобы запретить GDB переносить строки вывода.

15.5 Числа

Вы всегда можете вводить в GDB числа в восьмеричной, десятичной, или шестнадцатеричной системе в соответствии с обычными соглашениями: восьмеричные числа начинаются с '`0`', десятичные числа оканчиваются на '`.`' и шестнадцатеричные числа начинаются с '`0x`'. Числа, которые не начинаются ни с одного из этих знаков, по умолчанию считаются десятичными; аналогично, отображение чисел по умолчанию—если не определен никакой конкретный формат—осуществляется по основанию 10. Командой `set radix` вы можете изменять основание, устанавливаемое по умолчанию для ввода и вывода.

`set input-radix` *основание*

Установить основание по умолчанию для числового ввода. Поддерживаемые варианты для *основания*—8, 10 или 16. Само *основание* должно быть определено либо недвусмысленно, либо с использованием текущего основания системы счисления по умолчанию; например, любая из команд

```
set radix 012
set radix 10.
set radix 0xa
```

устанавливает десятичное основание. С другой стороны, '`set radix 10`' оставляет основание системы счисления без изменений независимо от того, каким оно было.

`set output-radix` *основание*

Установить основание по умолчанию для числового вывода. Поддерживаемые варианты для *основания*—8, 10 или 16. Само *основание* должно быть определено либо недвусмысленно, либо с использованием текущего основания системы счисления по умолчанию.

`show input-radix`
: Отобразить текущее основание по умолчанию для числового ввода.

`show output-radix`
: Отобразить текущее основание по умолчанию для числового вывода.

15.6 Необязательные предупреждения и сообщения

По умолчанию, GDB ничего не сообщает о своей внутренней деятельности. Если вы работаете на медленной машине, то вы можете захотеть использовать команду `set verbose`. Она велит GDB сообщать вам, когда он выполняет длинную внутреннюю операцию, чтобы вы не думали, что он завис.

В настоящее время, `set verbose` управляет только сообщениями о чтении таблиц символов исходного файла; смотрите Раздел 12.1 [Команды для задания файлов], с. 105, описание `symbol-file`.

`set verbose on`
: Разрешает GDB выводить определенные информационные сообщения.

`set verbose off`
: Отключает вывод GDB определенных информационных сообщений.

`show verbose`
: Сообщает, установлено `set verbose` в `on` или `off`.

По умолчанию, если GDB сталкивается с ошибками в таблице символов объектного файла, он не сообщает об этом; но если вы отлаживаете компилятор, эта информация может вам пригодиться (см. Раздел 12.2 [Ошибки чтения файлов с символами], с. 108).

`set complaints` *предел*
: Позволяет GDB выводить *предел* сообщений о каждом типе необычных символов прежде, чем перестать сообщать о проблеме. Установите *предел* в ноль для подавления всех сообщений, или очень большим, чтобы предотвратить подавление сообщений.

`show complaints`
: Выводит ограничение GDB на вывод сообщений о символах.

По умолчанию GDB осмотрителен, и задает, как иногда кажется, множество глупых вопросов, чтобы подтвердить определенные команды. Например, если вы пытаетесь выполнить программу, которая уже выполняется:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

Если вы неуклонно желаете сами разбираться с последствиями ваших собственных команд, вы можете отключить эту “возможность”:

`set confirm off`
: Отключает запросы подтверждений.

`set confirm on`
: Включает запросы подтверждений (по умолчанию).

`show confirm`
: Показывает, выводятся ли запросы подтверждений.

15.7 Необязательные сообщения о внутренних событиях

`set debug arch`
: Включает и отключает вывод отладочной информации gdbarch. По умолчанию отключено.

`show debug arch`
: Отображает текущее состояние вывода отладочной информации gdbarch.

`set debug event`
: Включает и отключает вывод отладочной информации о событиях GDB. По умолчанию отключено.

`show debug event`
: Отображает текущее состояние вывода отладочной информации о событиях GDB.

`set debug expression`
: Включает и отключает вывод отладочной информации о выражениях GDB. По умолчанию отключено.

`show debug expression`
: Отображает текущее состояние вывода отладочной информации о выражениях GDB.

`set debug overload`
: Включает и выключает вывод GDB отладочной информации о перегруженных символах Си++. Это включает такую информацию, как упорядочивание функций, и так далее. По умолчанию отключено.

`show debug overload`
: Отображает текущее состояние вывода GDB отладочной информации о перегруженных символах Си++.

`set debug remote`
: Включает и выключает вывод отчета о всех пакетах, посланных вперед и назад по последовательной линии удаленной машине. Информация печатается в стандартный выходной поток GDB. По умолчанию отключено.

`show debug remote`
: Выводит состояние вывода удаленных пакетов.

`set debug serial`
: Включает и выключает вывод GDB отладочной информации о последовательном соединении. По умолчанию отключено.

`show debug serial`
: Отображает текущее состояние вывода GDB отладочной информации о последовательном соединении.

`set debug target`
: Включает и выключает вывод GDB отладочной информации о цели. Эта включает информацию о том, что происходит на уровне цели GDB. По умолчанию отключено.

`show debug target`
: Отображает текущее состояние вывода GDB отладочной информации о цели.

`set debug varobj`
: Включает и отключает вывод GDB отладочной информации о переменных объектах. По умолчанию отключено.

`show debug varobj`
: Отображает текущее состояние вывода GDB отладочной информации о переменных объектах.

16 Фиксированные последовательности команд

Кроме команд точки останова (см. Раздел 5.1.7 [Команды точки останова], с. 41), GDB предоставляет два способа сохранить последовательности команд для выполнения целиком: определяемые пользователем команды и командные файлы.

16.1 Команды, определяемые пользователем

Команда, определяемая пользователем—это последовательность команд GDB, которой вы назначаете имя, как новой команде. Это осуществляется командой `define`. Пользовательские команды могут иметь до 10 параметров, разделенных пробелами. Внутри команды пользователя, доступ к параметрам производится посредством *$arg0. . . $arg9*. Вот простой пример:

```
define adder
  print $arg0 + $arg1 + $arg2
```

Для выполнения команды используйте:

```
adder 1 2 3
```

Этот пример определяет команду `adder`, которая печатает сумму трех своих параметров. Обратите внимание, что параметры являются текстовыми подстановками, так что они могут ссылаться на переменные, использовать сложные выражения или даже выполнять вызовы подчиненных функций.

`define` *имя-команды*
: Определить команду с именем *имя-команды*. Если команда с таким именем уже имеется, у вас запрашивается подтверждение на ее переопределение.

  Определение команды состоит из других командных строк GDB, которые задаются после команды `define`. Конец этих команд отмечается строкой, содержащей `end`.

`if`
: Имеет один аргумент—вычисляемое выражение. Команда сопровождается последовательностью команд, которые выполняются, если выражение истинно (отлично от нуля). Затем может следовать необязательная строка `else`, сопровождаемая последовательностью команд, которые выполняются только при ложном значении данного выражения. Конец списка отмечается строкой, содержащей `end`.

`while`
: Синтаксис подобен `if`: команда имеет один параметр, который является вычисляемым выражением и должен сопровождаться командами, по одной в строке, которые завершаются `end`. Выполнение команд повторяется, пока выражение истинно.

`document` *имя-команды*
: Описывает определенную пользователем команду *имя-команды*, так, чтобы к ней можно было обращаться посредством `help`. Команда *имя-команды* должна быть определена ранее. Эта команда считывает строки документации точно так же, как `define` считывает строки определения команды, до строки `end`. После завершения команды `document`, написанная вами документация будет отображаться командой `help` для команды *имя-команды*.

  Вы можете использовать команду `document` неоднократно, чтобы изменить документацию команды. Переопределение команды посредством `define` не изменяет документации.

`help user-defined`
: Перечислить все определенные пользователем команды, вместе с первой строкой документации (если она есть).

`show user`
`show user` *имя-команды*
: Вывести команды GDB, использовавшиеся для определения *имя-команды* (но не ее документацию). Если *имя-команды* не задано, вывести содержимое всех определенных пользователем команд.

При выполнении команд, определенных пользователем, команды определения не печатаются. Ошибка в любой из них останавливает выполнение всей определенной пользователем команды.

При использовании в интерактивном режиме, команды, обычно запрашивающие подтверждение, выполняются без запроса, если они используется внутри определенной пользователем команды. Многие команды GDB, которые обычно печатают сообщения о своих действиях, опускают их при использовании в команде, определенной пользователем.

16.2 Определяемые пользователем команды-ловушки

Вы можете определять *ловушки*, которые являются специальным видом определяемых пользователем команд. Всякий раз, когда вы выполняете команду '`foo`', перед ней выполняется определенная пользователем команда '`hook-foo`' (без параметров), если она существует.

Кроме того, существует псевдокоманда '`stop`'. Определение ('`hook-stop`') велит выполняться связанным с ней командам при каждом останове вашей программы: перед выполнением команд точек останова, перед выводом на экран сообщений или кадров стека.

Например, чтобы игнорировать сигналы `SIGALRM` во время выполнения в пошаговом режиме, но обрабатывать их при нормальном выполнении, вы можете определить:

```
define hook-stop
handle SIGALRM nopass
end

define hook-run
handle SIGALRM pass
end

define hook-continue
handle SIGLARM pass
end
```

Вы можете определить ловушку для любой однословной команды GDB, но не для синонимов команды; вам следует определить ловушку для базового имени команды, например, `backtrace`, но не `bt`. Если во время выполнения вашей ловушки возникает ошибка, выполнение команд GDB останавливается, и он выдает приглашение (до того, как введенная вами команда начнет выполняться).

Если вы попытаетесь определить ловушку, не соответствующую никакой известной команде, вы получите предупреждение от команды `define`.

16.3 Командные файлы

Командный файл для GDB—это файл, состоящий из строк с командами GDB. Такие файлы могут также включать комментарии (строки, начинающиеся с `#`). Пустая строка в командном файле ничего не делает; она не означает повторение последней команды, как это было бы при вводе с терминала.

Когда вы вызываете GDB, он автоматически выполняет команды из своих *файлов инициализации*. Это файлы, называющиеся '`.gdbinit`' в Unix и '`gdb.ini`' в DOS/Windows. Во время старта, GDB делает следующее:

1. считывает файл инициализации (если он существует) в вашем домашнем каталоге[1].
2. Обрабатывает ключи и операнды командной строки.
3. Считывает файл инициализации (если он существует) в текущем рабочем каталоге.
4. Считывает командные файлы, заданные с помощью ключа '`-x`'.

Файл инициализации в вашем домашнем каталоге может устанавливать параметры (такие как '`set complaints`'), которые влияют на последующую обработку ключей и операндов командной строки. Файлы инициализации не выполняются, если вы используете ключ '`-nx`' см. Раздел 2.1.2 [Выбор режимов], с. 11.

В некоторых конфигурациях GDB, файлу инициализации присваивается другое имя (обычно это среды, где специализированная форма GDB должна сосуществовать с другими формами, следовательно должно быть отличное имя для файла инициализации специализированной версии). Следующие среды используют специальные имена файлов инициализации:

- VxWorks (ОС реального времени Wind River Systems): '`.vxgdbinit`'
- OS68K (ОС реального времени Enea Data Systems): '`.os68gdbinit`'
- ES-1800 (эмулятор Ericsson Telecom AB M68000): '`.esgdbinit`'

Вы также можете запросить выполнение командного файла с помощью команды `source`:

`source` *имя-файла*
: Выполнить командный файл *имя-файла*.

Строки командного файла выполняются последовательно, при этом они не выводятся. Ошибка в любой команде завершает выполнение всего командного файла.

Команды, запрашивающие подтверждение в интерактивном режиме, при выполнении в командном файле выполняются без запросов. Многие команды GDB, обычно выводящие сообщения о своих действиях, опускают эти сообщения при вызове из командных файлов.

16.4 Команды для управляемого вывода

Во время выполнения командного файла или определенной пользователем команды, нормальный вывод GDB подавляется; единственый появляющийся вывод—тот, который производится явно командами из определения. В этом разделе описываются три команды, полезные для получения именно такого вывода, который вы хотите.

`echo` *текст*
: Напечатать *текст*. Непечатные знаки могут быть включены в *текст* при помощи экранирующих последовательностей Си, таких как '`\n`' для перехода на новую строку. **Переход на новую строку не печатается, если вы его не укажете.** В дополнение к стандартным экранирующим последовательностям Си, обратная косая черта, за которой следует пробел, обозначает пробел. Это полезно для отображения строки с пробелами в начале или конце, поскольку в противном случае начальные и конечные пробелы удаляются из всех аргументов. Чтобы напечатать ' `and foo =` ', используйте команду '`echo \ and foo = \ `'.

  Обратная косая черта в конце *текста* может использоваться, как и в Си, для продолжения команды на последующие строки. Например,

[1] В системах DOS/Windows, домашним каталогом считается тот, который указывает переменная среды `HOME`.

```
echo Вот пример текста,\n\
который занимает\n\
несколько строк.\n
```

производит такой же вывод как

```
echo Вот пример текста,\n
echo который занимает\n
echo несколько строк.\n
```

`output` *выражение*
: Напечатать значение *выражения* и ничего кроме него: никаких новых строк, никаких ‘$*nn* = ’. Значение также не заносится в историю значений. См. Раздел 8.1 [Выражения], с. 61, для дополнительной информации о выражениях.

`output/`*формат выражение*
: Вывести значение *выражения* в формате *формат*. Вы можете использовать те же форматы, что и для `print`. См. Раздел 8.4 [Форматы вывода], с. 64, для получения большей информации.

`printf` *строка*`,` *выражения*`...`
: Напечатать значения *выражений*, причем формат вывода задает *строка*. *Выражения* разделяются запятыми и могут быть либо числами, либо указателями. Их значения печатаются так, как определяет *строка*, в точности, как если бы ваша программа выполняла подпрограмму Си

  ```
  printf (строка, выражения...);
  ```

  Например, вы можете напечатать два шестнадцатеричных значения:

  ```
  printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
  ```

  Единственые экранирующие последовательности с обратной косой чертой, которые вы можете использовать в строке формата—простые последовательности, состоящие из обратной косой черты, за которой следует буква.

17 Использование GDB под управлением GNU Emacs

Специальный интерфейс позволяет вам использовать GNU Emacs для просмотра (и редактирования) исходных файлов программы, которую вы отлаживаете с помощью GDB.

Чтобы использовать этот интерфейс, используйте команду *M-x gdb* в Emacs. В качестве аргумента задайте выполняемый файл, который вы хотите отладить. Эта команда вызывает GDB как подпроцесс Emacs с вводом и выводом через заново созданный буфер.

Под управлением Emacs, GDB используется точно так же, как обычно, за исключением двух моментов:

- Весь "терминальный" ввод и вывод происходит через буфер Emacs.

Это относится как к командам GDB и их выводу, так и к вводу и выводу, производимыми отлаживаемой программой.

Это полезно, потому что вы можете копировать текст предыдущих команд и вводить их снова; вы даже можете использовать таким образом части вывода.

Все средства режима оболочки Emacs доступны для взаимодействия с вашей программой. В частности, вы можете посылать сигналы обычным путем— например, *C-c C-c* для прерывания, *C-c C-z* для остановки.

- GDB отображает исходный код через Emacs.

Каждый раз, когда GDB отображает кадр стека, Emacs автоматически находит исходный файл для него и помещает стрелку ('`=>`') на левом крае текущей строки. Emacs использует отдельный буфер для отображения исходного текста, и разделяет экран, чтобы отобразить как сеанс GDB, так и исходный текст.

Тем не менее, явные команды GDB `list` и `search` производят вывод как обычно, но у вас вероятно не будет причин использовать их из Emacs.

> *Предупреждение:* Если каталог, в котором находится ваша программа, не является текущим, Emacs легко может ошибиться при определении местонахождения исходных файлов; в этом случае вспомогательный буфер с исходным текстом не появляется. GDB может искать программы, используя переменную среды `PATH`, так что сеансы ввода и вывода происходят нормально; но Emacs не получает достаточно информации от GDB, чтобы найти исходные файлы в такой ситуации. Чтобы избежать этой проблемы, либо запускайте режим GDB из каталога, где находится ваша программа, либо укажите абсолютное имя файла, когда будет запрошен параметр для *M-x gdb*.
>
> Подобная путаница может возникнуть, если вы используете команду GDB `file`, чтобы переключиться к отладке программы, находящейся в каком-нибудь другом месте, из существующего буфера GDB в Emacs.

По умолчанию, *M-x gdb* вызывает программу с именем '`gdb`'. Если вам нужно вызвать GDB под другим именем (например, если вы храните несколько конфигураций под различными именами), вы можете установить переменную Emacs `gdb-command-name`; например, установка

```
(setq gdb-command-name "mygdb")
```

(которой предшествует *M-:* или *ESC :*, или если она введена в буфер `*scratch*` или в вашем файле '`.emacs`'), заставит Emacs вызвать программу "`mygdb`".

В буфере ввода-вывода GDB, вы можете использовать следующие специальные команды Emacs в дополнение к стандартным командам режима оболочки:

C-h m Описывает возможности режима GDB Emacs.

M-s Выполнить до другой строки исходного текста, подобно команде GDB `step`; также обновляет окно отображения для показа текущего файла и положения в нем.

M-n Выполнить до следующей строки исходного текста в этой функции, пропуская все вызовы функций, подобно команде GDB `next`. Затем обновить окно отображения, чтобы показать текущий файл и положение в нем.

M-i Выполнить одну инструкцию, подобно команде GDB `stepi`; обновляет окно отображения.

M-x gdb-nexti
Выполнить до следующей инструкции, используя команду GDB `nexti`; обновляет окно отображения.

C-c C-f Выполнить до выхода из выбранного кадра стека, подобно команде GDB `finish`.

M-c Продолжить выполнение вашей программы, подобно команде GDB `continue`.

Предупреждение: В Emacs версии 19, эта команда вызывается посредством *C-c C-p*.

M-u Подняться вверх по стеку на число кадров, заданное числовым параметром, подобно команде GDB `up` (см. раздел “Числовые аргументы” в *Руководство по* GNU *Emacs*).

Предупреждение: В Emacs версии 19, эта команда вызывается посредством *C-c C-u*.

M-d Спуститься вниз по стеку на число кадров, заданное числовым параметром, аналогично команде GDB `down`.

Предупреждение: В Emacs версии 19, эта команда вызывается посредством *C-c C-d*.

C-x & Прочитать число в позиции курсора и вставить его в конец буфера ввода-вывода GDB. Например, если вы хотите дисассемблировать код по адресу, отображенному ранее, наберите *disassemble*; затем переместите курсор на изображение адреса и задайте аргумент для `disassemble`, набрав *C-x &*.

Вы можете и дальше настраивать это, определяя элементы списка `gdb-print-command`; если он определен, вы можете форматировать или как-либо иначе обрабатывать числа, задаваемые *C-x &*, перед их вставкой в буфер. Числовой аргумент к *C-x &* показывает, что вам нужно специальное форматирование, а также действует как индекс для выбора элемента из списка. Если элемент списка является строкой, то вставляемое число форматируется с использованием функции Emacs `format`; в противном случае, число передается как параметр соответствующему элементу списка.

В любом исходном файле, команда Emacs *C-x SPC* (`gdb-break`) велит GDB установить точку останова на строке исходного текста, в которой находится точка.

Если вы случайно удалите буфер отображения исходного текста, его можно легко вернуть обратно, введя команду `f` в буфере GDB, чтобы запросить отображение кадра; когда выполнение происходит под управлением Emacs, это при необходимости создает исходный буфер заново, чтобы показать вам текущий кадр.

Исходные файлы, отображаемые в Emacs, находятся в обычных буферах Emacs, которые получают доступ к исходным файлам обычным способом. При желании вы можете редактировать файлы в этих буферах; но помните, что GDB взаимодействует с Emacs в терминах номеров строк. Если вы добавите или удалите строки из текста, известные GDB номера строк больше не будут соответствовать коду.

18 Примечания GDB

Эта глава описывает примечания в GDB. Примечания разработаны для согласования GDB с графическими интерфейсами пользователя или другими аналогичными программами, которые хотят взаимодействовать с GDB на относительно высоком уровне.

18.1 Что такое примечание?

Чтобы создавать примечания, запустите GDB с ключем `--annotate=2`.

Примечания начинаются с символа новой строки, двух символов ‘`control-z`’ и имени примечания. Если нет дополнительной информации, связанной с примечанием, непосредственно за его именем следует символ новой строки. Если дополнительная информация есть, за именем примечания следует пропуск, дополнительная информация и символ новой строки. Дополнительная информация не может содержать символа новой строки.

Любой вывод, не начинающийся с символа новой строки и двух ‘`control-z`’, означает буквальный вывод GDB. В настоящее время GDB не нужно выводить два ‘`control-z`’ вслед за символом новой строки, но если это понадобится, примечания могут быть расширены ‘экранирующим’ примечанием, которое означает вывод этих трех символов.

Вот простой пример запуска GDB с примечаниями:

```
$ gdb -annotate=2
GNU GDB 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "sparc-sun-sunos4.1.3"

^Z^Zpre-prompt
(gdb)
^Z^Zprompt
quit

^Z^Zpost-prompt
$
```

Здесь ‘`quit`’ является для GDB вводом; остальное—вывод GDB. Три строки, начинающиеся с ‘`^Z^Z`’ (где ‘`^Z`’ означает знак ‘`control-z`’), суть примечания; остальное является выводом GDB.

18.2 Префикс server

Чтобы подать команду GDB, не оказывая влияния на определенные аспекты состояния, видимые для пользователей, начните ее с ‘`server `’. Это означает, что данная команда не воздействует на историю команд, а также не влияет на понятие GDB о том, какую команду повторять, если в пустой строке нажата клавиша ⟨RET⟩.

Префикс server не влияет на запись значений в историю значений; чтобы напечатать значение, не занося его в историю, используйте команду `output` вместо `print`.

18.3 Значения

Когда значение выводится в различных контекстах, GDB использует примечания, чтобы отделить его от окружающего текста.

Если значение выводится с помощью `print` и добавляется в историю значений, примечание выглядит так:

```
^Z^Zvalue-history-begin номер-в-истории флаги-значения
строка-истории
^Z^Zvalue-history-value
значение
^Z^Zvalue-history-end
```

где *номер-в-истории*—номер, который значение получает в истории, *строка-истории*—строка, такая как '`$5 = `', которая представляет значение пользователю, *значение* является выводом, соответствующим самому значению, а *флаги-значения*—'`*`' для значения, которое может быть разыменовано, и '`-`', если нет.

Если значение не добавляется в историю значений (это может быть или неверное число с плавающей точкой, или оно выводится командой `output`), примечание выглядит похожим образом:

```
^Z^Zvalue-begin флаги-значения
значение
^Z^Zvalue-end
```

Когда GDB выводит аргумент функции (например, в выводе команды `backtrace`), он делает такие примечания:

```
^Z^Zarg-begin
имя-аргумента
^Z^Zarg-name-end
строка-разделитель
^Z^Zarg-value флаги-значения
значение
^Z^Zarg-end
```

где *имя-аргумента* есть имя аргумента, *строка-разделитель*—текст (такой как '`=`'), который отделяет имя от значения для удобства пользователя, а *флаги-значения* и *значение* имеют такой же смысл, что и в примечании `value-history-begin`.

При выводе структуры, GDB делает следующие примечания:

```
^Z^Zfield-begin флаги-значения
имя-поля
^Z^Zfield-name-end
строка-разделитель
^Z^Zfield-value
значение
^Z^Zfield-end
```

где *имя-поля* есть имя поля, *строка-разделитель*—текст (такой как '`=`'), который отделяет имя от значения для удобства пользователя, а *флаги-значения* и *значение* имеют тот же смысл, что и в примечании `value-history-begin`.

При выводе массива, GDB делает следующие примечания:

```
^Z^Zarray-section-begin индекс-в-массиве флаги-значения
```

где *индекс-в-массиве*—индекс первого аннотируемого элемента, а *флаги-значения* имеют такой же смысл, что и в примечании `value-history-begin`. За этим следует произвольное число элементов. Элемент может быть либо одиночным

‘,’ *пропуск* ; опускается для первого элемента
значение
^Z^Zelt

либо повторяющимся

‘,’ *пропуск* ; опускается для первого элемента
значение
^Z^Zelt-rep *число-повторений*
строка-повторений
^Z^Zelt-rep-end

В обоих случаях, *значение* является выводом значения элемента, а *пропуск* может содержать пробелы, символы табуляции и новой строки. В случае повторяющихся элементов, *число-повторений* представляет число последовательных элементов массива, которые содержат данное значение, а *строка-повторений* является строкой, которая предназначена для уведомления пользователя о том, что выводятся повторяющиеся элементы.

После того, как выведены все элементы массива, примечание к массиву заканчивается так:

^Z^Zarray-section-end

18.4 Кадры

Когда GDB печатает кадр, он делает к нему примечания. Например, это применяется к кадрам, выводимым при остановке GDB, в результате вывода командами, такими как `backtrace` или `up`, и так далее.

Примечания к кадру начинаются с

^Z^Zframe-begin *уровень адрес*
строка-уровня

где *уровень*—это номер кадра (0 для самого внутреннего кадра, другие кадры имеют положительные номера), *адрес*—это адрес кода, выполняющегося в данном кадре, а *строка-уровня*—строка, предназначенная для передачи уровня пользователю. *Адрес* имеет форму ‘`0x`’, за которым следует одна или более шестнадцатеричных цифр в нижнем регистре (заметьте, что это не зависит от языка). Кадр заканчивается так:

^Z^Zframe-end

Между этими комментариями находится основное тело кадра, которое может состоять из

- ^Z^Zfunction-call
  строка-вызова-функции

  где *строка-вызова-функции* является текстом, предназначенным для уведомления пользователя, что этот кадр связан с вызовом функции, который GDB сделал в отлаживаемой программе.

- ^Z^Zsignal-handler-caller
  строка-вызова-обработчика-сигнала

  где *строка-вызова-обработчика-сигнала*—текст, предназначенный для уведомления пользователя, что этот кадр связан с тем механизмом, который использовался операционной системой при вызове обработчика сигнала (это тот кадр, из которого произошел вызов обработчика, а не кадр для самого обработчика).

- Обычный кадр.

  Это может, возможно (в зависимости от того, считается ли это информацией, интересной для пользователя), начинаться с

  ```
  ^Z^Zframe-address
  адрес
  ^Z^Zframe-address-end
  строка-разделитель
  ```

  здесь *адрес*—это адрес, где происходит выполнение в кадре (тот же адрес, что и в примечании `frame-begin`, но выведенный в форме, предназначенной для пользователя—в частности, синтаксис различается в зависимости от языка), а *строка-разделитель* является строкой, предназначенной для отделения этого адреса от того, что за ним следует для удобства пользователя.

  Затем идет

  ```
  ^Z^Zframe-function-name
  имя-функции
  ^Z^Zframe-args
  аргументы
  ```

  где *имя-функции* есть имя функции, выполняющейся в кадре, или '`??`', если оно не известно, а *аргументы*—это аргументы к кадру, со скобками вокруг них (каждый аргумент аннотируется также индивидуально, см. Раздел 18.3 [Примечания к значениям], с. 164).

  Если доступна информация об исходных текстах, печатается ссылка на них:

  ```
  ^Z^Zframe-source-begin
  вводная-исходная-строка
  ^Z^Zframe-source-file
  имя-файла
  ^Z^Zframe-source-file-end
  :
  ^Z^Zframe-source-line
  номер-строки
  ^Z^Zframe-source-end
  ```

  где *вводная-исходная-строка* отделяет ссылку от предшествующего ей текста, для удобства пользователя, *имя-файла*—это имя исходного файла, *номер-строки*—номер строки в этом файле (первая строка имеет номер 1).

  Если GDB печатает некоторую информацию о том, откуда появился этот кадр (какая библиотека, какой сегмент загрузки, и так далее; в настоящее время реализовано только на RS/6000), он делает такие примечания:

  ```
  ^Z^Zframe-where
  информация
  ```

  Затем, если исходный текст действительно должен быть отображен для этого кадра (например, это не верно для вывода от команды `backtrace`), тогда выводится примечание `source` (см. Раздел 18.11 [Примечания к исходному тексту], с. 170). В отличие от большинства примечаний, этот вывод производится вместо обычного текста, который был бы напечатан, а не в дополнение к нему.

18.5 Отображения

Когда GDB велят отобразить что-то с помощью команды `display`, к результату отображения делаются примечания:

```
^Z^Zdisplay-begin
```
номер
```
^Z^Zdisplay-number-end
```
разделитель-номеров
```
^Z^Zdisplay-format
```
формат
```
^Z^Zdisplay-expression
```
выражение
```
^Z^Zdisplay-expression-end
```
разделитель-выражений
```
^Z^Zdisplay-value
```
значение
```
^Z^Zdisplay-end
```

здесь *номер*—это номер отображения, *разделитель-номеров* предназначен для отделения номеров от того, что следует затем для пользователя, *формат* включает информацию о том, как отображается значение, такую как размер, формат и так далее, *выражение*—это отображаемое выражение, *разделитель-выражений* предназначен для отделения выражения от следующего за ним текста для пользователя, и *значение*—это действительное значение, которое отображается.

18.6 Примечания ко вводу GDB

Когда GDB выводит приглашение для ввода, он делает к этому примечания, так что становится возможным узнать, когда посылать данные, когда закончен вывод от данной команды, и так далее.

Каждый из различных видов ввода имеет различный *тип ввода*. Каждый тип ввода имеет три примечания: примечание `pre-`, обозначающее начало каждого выводимого приглашения, простое примечание, обозначающее конец приглашения, и затем примечание `post-`, обозначающее конец любого эхо, которое может быть ассоциировано (а может и не быть) со вводом. Например, характерной чертой типа ввода `prompt` являются следующие примечания:

```
^Z^Zpre-prompt
^Z^Zprompt
^Z^Zpost-prompt
```

Существуют следующие типы ввода:

`prompt` Когда GDB запрашивает команду (главное приглашение GDB).

`commands` Когда GDB запрашивает набор команд, как в команде `commands`. Примечания повторяются для каждой введенной команды.

`overload-choice`
Когда GDB хочет, чтобы пользователь выбрал одну из нескольких перегруженных функций.

`query` Когда GDB хочет, чтобы пользователь подтвердил потенциально опасное действие.

`prompt-for-continue`
Когда GDB запрашивает у пользователя нажатие ввода для продолжения. Замечание: не ожидайте, что это будет работать хорошо; используйте вместо этого `set height 0` для отключения приглашений. Это происходит потому, что при наличии примечаний подсчет строк происходит неверно.

18.7 Ошибки

```
^Z^Zquit
```

Это примечание появляется непосредственно перед тем, как GDB отвечает на прерывание.

```
^Z^Zerror
```

Это примечание появляется сразу перед тем, как GDB отвечает на ошибку.

Примечания выхода и ошибки обозначают, что любое примечание, в середине которого находился GDB, могут внезапно оборваться. Например, если за примечанием `value-history-begin` следует `error`, то не нужно ожидать соответствующий `value-history-end`. Однако, не следует также ожидать, что его точно не будет; примечание об ошибке не обязательно означает, что GDB немедленно возвращается в начало на самый верхний уровень.

Примечанию к ошибке или выходу может предшествовать

```
^Z^Zerror-begin
```

Весь вывод между этим и примечанием к ошибке или выходу является сообщением об ошибке.

Пока примечаний к предупреждающим сообщениям не делается.

18.8 Информация о точке останова

К выводу, производимому командой `info breakpoints`, делаются следующие примечания:

```
^Z^Zbreakpoints-headers
элемент-заголовка
^Z^Zbreakpoints-table
```

где *элемент-заголовка* имеет тот же синтаксис, что и элемент (смотрите ниже), но вместо данных, он содержит строки, которые предназначены для разъяснения пользователю значений каждого поля. Затем следует произвольное число элементов. Если поле не подходит к этому элементу, оно опускается. Поля могут содержать завершающие пропуски. Каждое поле состоит из:

```
^Z^Zrecord
^Z^Zfield 0
номер
^Z^Zfield 1
тип
^Z^Zfield 2
положение
^Z^Zfield 3
включена
^Z^Zfield 4
адрес
^Z^Zfield 5
что
^Z^Zfield 6
кадр
^Z^Zfield 7
условие
^Z^Zfield 8
счетчик-игнорирований
^Z^Zfield 9
```

команды

Заметьте, что *адрес* предназначен для использования пользователем—синтаксис различается в зависимости от языка.

Вывод заканчивается так:

```
^Z^Zbreakpoints-table-end
```

18.9 Сообщения о недостоверности

Следующие примечания говорят о том, что определенные куски информации, описывающие состояние, могли измениться.

`^Z^Zframes-invalid`
: Кадры (например, вывод команды `backtrace`) могли измениться.

`^Z^Zbreakpoints-invalid`
: Точки останова могли измениться. Например, пользователь только что добавил или удалил точку останова.

18.10 Выполнение программы

Когда программа начинает выполняться вследствие команды GDB, такой как `step` или `continue`, выводится

```
^Z^Zstarting
```

Когда программа останавливается, выводится

```
^Z^Zstopped
```

Перед примечанием `stopped`, множество примечаний описывают, как программа остановилась.

`^Z^Zexited` *код-выхода*
: Программа завершилась, и *код-выхода* является кодом выхода (ноль при успешном завершении, в противном случае не ноль).

`^Z^Zsignalled`
: Программа завершилась по сигналу. После `^Z^Zsignalled`, примечания продолжаются:

```
вступительный-текст
^Z^Zsignal-name
имя
^Z^Zsignal-name-end
текст-в-середине
^Z^Zsignal-string
строка
^Z^Zsignal-string-end
заключительный-текст
```

: где *имя* является именем сигнала, таким как `SIGILL` или `SIGSEGV`, а *строка* представляет объяснение сигнала, такое как `Illegal Instruction` или `Segmentation fault`. *Вступительный-текст*, *текст-в-середине* и *заключительный текст* используются для удобства пользователя и не имеют определенного формата.

`^Z^Zsignal`
: Синтаксис этого примечания такой же, как для `signalled`, но GDB сообщает, что программа лишь получила сигнал, а не то, что она остановилась из-за него.

`^Z^Zbreakpoint` *номер*
: Программа достигла точки останова с номером *номер*.

`^Z^Zwatchpoint` *номер*
: Программа достигла точки наблюдения с номером *номер*.

18.11 Вывод исходного текста

Следующие примечания используются вместо вывода исходного текста:

`^Z^Zsource` *имя-файла*:*строка*:*символ*:*middle*:*адрес*

где *имя-файла* указывает абсолютное имя файла, *строка*—это номер строки в этом файле (первая строка имеет номер 1), *символ*—позиция символа в файле (первый символ в исходном файле имеет номер 0) (для большинства отладочных форматов это будет обязательно указывать на начало строки), *middle* есть '`middle`', если *адрес* находится в середине строки, или '`beg`', если *адрес* находится в начале строки, а *адрес* является адресом в целевой программе, ассоциированным с выводимым исходным текстом. *Адрес* записан в форме '`0x`', за которым следует одна или несколько шестнадцатеричных цифр в нижнем регистре (заметьте, что это не зависит от языка).

18.12 Примечания, которые могут понадобиться с будущем

- target-invalid
  цель могла измениться (регистры, содержимое памяти или статус выполнения). Для эффективности выполнения, мы можем захотеть определять 'register-invalid' и 'all-register-invalid' с большей точностью

- систематические примечания к параметрам set/show (включая сообщения о недостоверности).

- аналогично, 'info' возвращает список кандидатов на сообщение о недостоверности.

19 Интерфейс GDB/MI

Назначение и цель

GDB/MI является построчным, машинно-ориентированным текстовым интерфейсом к GDB. Он предназначен специально для поддержки разработки систем, которые используют отладчик лишь как одну маленькую компоненту большой системы.

Эта глава является спецификацией интерфейса GDB/MI. Она написана в форме справочного руководства.

Заметьте, что GDB/MI все еще находится в стадии разработки, так что некоторые описанные ниже возможности являются неполными и могут быть изменены.

Система обозначений и терминология

Эта глава использует следующую систему обозначений:

- `|` разделяет две альтернативы.
- `[` *нечто* `]` указывает, что *нечто* является необязательным: оно может быть задано, а может и нет.
- `(` *группа* `)*` означает, что *группа* в скобках может повторяться ноль и более раз.
- `(` *группа* `)+` означает, что *группа* в скобках может повторяться один и более раз.
- `"`*строка*`"` обозначает текст *строка*.

Выражения признательности

В алфавитном порядке: Елена Заннони, Эндрю Кагни, Фернандо Нассер и Стан Шебс.

19.1 Синтаксис команд GDB/MI

19.1.1 Входной синтаксис GDB/MI

команда ↦
: *команда-cli* `|` *команда-mi*

команда-cli ↦
: `[` *лексема* `]` *команда-cli nl*, где *команда-cli* есть любая существующая команда GDB CLI.

команда-mi ↦
: `[` *лексема* `]` `"-"` *действие* `(` `" "` *ключ* `)*` `[` `" --"` `]` `(` `" "` *параметр* `)*` *nl*

лексема ↦
: "любая последовательность цифр"

ключ ↦ `"-"` *параметр* `[` `" "` *параметр* `]`

параметр ↦
: *непустая-последовательность* `|` *строка-си*

действие ↦
: *любое из действий, описанных в этой главе*

непустая-последовательность ↦
все что угодно, не содержащее специальных знаков, таких как `"-"`*, nl,* `"""` *и, конечно,* `" "`

строка-си ↦
`"""` *содержимое-семибитной-строки-iso-си* `"""`

nl ↦ `CR | CR-LF`

Замечания:

- Команды CLI все еще обрабатываются интерпретатором MI; их вывод описан ниже.
- *Лексема*, если присутствует, передается назад, когда выполнение команды завершается.
- Некоторые команды MI допускают необязательные аргументы как часть списка параметров. Каждый ключ идентифицируется предшествующей ему чертой ‘`-`’, и за ним может следовать в качестве параметра необязательный аргумент. Ключи появляются в начале списка параметров и могут быть отделены от обычных параметров при помощи ‘`--`’ (это полезно, когда некоторые параметры начинаются с черты).

Прагматические соображения:

- Мы хотим получить простой доступ к существующему синтаксису CLI (для отладки).
- Мы хотим, чтобы работа MI была легко заметна.

19.1.2 Выходной синтаксис GDB/MI

Вывод GDB/MI состоит из нуля или более внеочередных записей, за которыми может следовать одна результирующая запись. Эта запись относится к самой последней команде. Последовательность выводимых записей завершается ‘`(gdb)`’.

Если входная команда начиналась с префикса *лексема*, то соответствующий вывод для этой команды также будет начинаться с того же префикса *лексема*.

вывод ↦ `(` *внеочередная-запись* `)*` `[` *результирующая-запись* `]` `"(gdb)"` *nl*

результирующая-запись ↦
`[` *лексема* `]` `"^"` *результирующий-класс* `(` `","` *результат* `)*` *nl*

внеочередная-запись ↦
асинхр-запись `|` *поточн-запись*

асинхр-запись ↦
асинхр-вывод-выполн `|` *асинхр-вывод-статуса* `|` *асинхр-вывод-уведомл*

асинхр-вывод-выполн ↦
`[` *лексема* `]` `"*"` *асинхр-вывод*

асинхр-вывод-статуса ↦
`[` *лексема* `]` `"+"` *асинхр-вывод*

асинхр-вывод-уведомл ↦
`[` *лексема* `]` `"="` *асинхр-вывод*

асинхр-вывод ↦
асинхр-класс `(` `","` *результат* `)*` *nl*

результ-класс ↦
`"done" | "running" | "connected" | "error" | "exit"`

асинхр-класс ↦
`"stopped"` `|` *другое* (где *другое* будет добавлено по необходимости—это все еще находится в стадии разработки).

результат ↦
    `[` *строка* `"="` `]` *значение*

значение ↦
    константа | `"{"` *результат* `(` `","` *результат* `)*` `"}"`

константа ↦
    строка-си

поточн-запись ↦
    консольн-поточн-вывод | *целев-поточн-вывод* | *журн-поточн-вывод*

консольн-поточн-вывод ↦
    `"~"` *строка-си*

целев-поточн-вывод ↦
    `"@"` *строка-си*

журн-поточн-вывод ↦
    `"&"` *строка-си*

nl ↦ `CR` | `CR-LF`

лексема ↦
    любая последовательность цифр.

Кроме того, следующее все еще находится в стадии разработки:

запрос Это действие пока не определено.

Замечания:

- Все выходные последовательности заканчиваются одной строкой, содержащей точку.
- *Лексема* берется из соответствующего запроса. Если выполнение команды прерывается командой '`-exec-interrupt`', *лексема*, ассоциированная с сообщением '`*stopped`', является лексемой исходной выполняемой команды, а не лексемой команды прерывания.
- *Асинхр-вывод-статуса* содержит последующую информацию состояния о выполнении медленной операции. Она может быть отброшена. Весь вывод состояния начинается с префикса '`+`'.
- *Асинхр-вывод-выполн* содержит асинхронное изменения состояния на цели (остановлена, запущена, исчезла). Весь асинхронный вывод начинается с префикса '`*`'.
- *Асинхр-вывод-уведомл* содержит сопровождающую информацию, которую должен обработать клиент (например, информацию о новой точке останова). Весь уведомительный вывод начинается с префикса '`=`'.
- *Консольн-поточн-вывод* является выводом, который должен быть отображен на консоли без изменений. Он является текстовым ответом на команду CLI. Весь консольный вывод начинается с префикса '`~`'.
- *Целев-поточн-вывод* является выводом, произведенным целевой программой. Весь целевой вывод начинается с префикса '`@`'.
- *Журн-поточн-вывод* является выходным текстом, происходящим от внутренней реализации GDB, например, сообщения, которые должны быть отображены как часть журнала ошибок. Весь журнальный вывод начинается с префикса '`&`'.

См. Раздел 19.3.2 [Поточные записи GDB/MI], с. 175, для более подробной информации о различных выводимых записях.

См. Раздел 19.15 [Черновик изменений к выходному синтаксису GDB/MI], с. 222, для предлагаемых изменений к выходному синтаксису.

19.1.3 Простые примеры взаимодействия с GDB/MI

Этот подраздел представляет несколько простых примеров взаимодействия с использованием интерфейса GDB/MI. В этих примерах, '`->`' означает, что следующая строка передается GDB/MI в качестве ввода, а '`<-`' означает вывод, полученный от GDB/MI.

Останов цели

Вот пример останова подчиненного процесса:

```
-> -stop
<- (gdb)
```

и затем:

```
<- *stop,reason="stop",address="0x123",source="a.c:123"
<- (gdb)
```

Простая команда CLI

Вот пример простой команды CLI, передаваемой ему через GDB/MI.

```
-> print 1+2
<- ~3\n
<- (gdb)
```

Команда с побочными эффектами

```
-> -symbol-file xyz.exe
<- *breakpoint,nr="3",address="0x123",source="a.c:123"
<- (gdb)
```

Плохая команда

Вот что происходит, если вы передаете несуществующую команду:

```
-> -rubbish
<- error,"Rubbish not found"
<- (gdb)
```

19.2 Совместимость GDB/MI с CLI

Чтобы помочь пользователям, знакомым с существующим в GDB интерфейсом CLI, GDB/MI принимает существующие команды CLI. Как определено синтаксисом, такие команды могут быть непосредственно введены в интерфейс GDB/MI, и GDB будет отвечать.

Этот механизм предоставляется для помощи разработчикам клиентов GDB/MI, а не как надежный интерфейс к CLI. Так как команда интерпретируется в среде, которая подразумевает поведение GDB/MI, точные результаты таких команд в конечном итоге скорее всего станут неудобоваримой смесью вывода GDB/MI и CLI.

19.3 Выходные записи GDB/MI

19.3.1 Результирующие записи GDB/MI

В дополнение к множеству внеочередных уведомлений, ответ на команду GDB/MI включает один из следующих указателей результата:

`"^done" [ "," `*результаты*` ]`
: Синхронная операция прошла успешно, возвращаемыми значениями являются *результаты*.

`"^running"`
: Асинхронная операция была успешно начата. Цель выполняется.

`"^error" "," `*строка-си*
: Операция завершилась с ошибкой. *Строка-си* содержит соответствующее сообщение об ошибке.

19.3.2 Поточные записи GDB/MI

GDB хранит число выходных потоков: консоль, цель и журнал. Вывод, предназначенный для этих потоков, пропускается через интерфейс GDB/MI при помощи *поточных записей*.

Каждая поточная запись начинается с уникального *префиксного символа*, который идентифицирует свой поток (см. Раздел 19.1.2 [Выходной синтаксис GDB/MI], с. 172). Помимо префикса, каждая поточная запись содержит *строку-вывод*. Это либо простой текст (с подразумеваемым знаком новой строки), или Си-строка в кавычках (которая не содержит подразумеваемого знака новой строки).

`"~" `*строка-вывод*
: Консольный поток вывода содержит текст, который должен быть отображен в консольном окне CLI. Он содержит текстовые ответы на команды CLI.

`"@" `*строка-вывод*
: Целевой поток вывода содержит произвольный текстовый вывод от выполняемой цели.

`"&" `*строка-вывод*
: Журнальный поток содержит отладочные сообщения, которые создает сам GDB.

19.3.3 Внеочередные записи GDB/MI

Внеочередные записи используются для уведомления клиента GDB/MI о произошедших дополнительных изменениях. Эти изменения могут либо исходить от GDB/MI (например, измененная точка останова), либо быть результатом действий цели (например, цель остановилась).

Ниже приведен предварительный список возможных внеочередных записей.

`"*" "stop"`

19.4 Формат описания команд GDB/MI

Оставшиеся разделы описывают блоки команд. Каждый блок команд схематично аналогичен этому разделу.

Заметьте, что разбиения строк в примерах присутствуют только для удобства чтения. Они не появляются в реальном выводе. Учтите также, что команды с недоступными примерами (Н.П.) еще не реализованы.

Мотивация

Мотивация для этого набора команд.

Введение

Краткое введение в этот набор команд в целом.

Команды

Для каждой команды в блоке, описано следующее:

Краткое описание

`-command` *арг...*

Команда GDB

Соответствующая команда CLI GDB.

Результат

Внеочередные сообщения

Примечания

Пример

19.5 Команды GDB/MI для таблицы точек останова

Этот раздел описывает команды GDB/MI для управления точками останова.

Команда `-break-after`

Краткое описание

`-break-after` *номер число*

Точка останова с номером *номер* не срабатывает, пока она не будет достигнута *число* раз. Чтобы увидеть, как это отражается на выводе команды '`-break-list`', смотрите ниже описание команды '`-break-list`'.

Команда GDB

Соответствующей командой GDB является '`ignore`'.

Пример

```
(gdb)
-break-insert main
^done,bkpt={number="1",addr="0x000100d0",file="hello.c",line="5"}
(gdb)
-break-after 1 3
~
^done
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0",
ignore="3"}}
(gdb)
```

Команда `-break-condition`

Краткое описание

`-break-condition` *номер выраж*

Точка останова *номер* остановит программу, только если условие *выраж* истино. Условие становится частью вывода '`-break-list`' (смотрите ниже описание команды '`-break-list`').

Команда GDB

Соответствующей командой GDB является '`condition`'.

Пример

```
(gdb)
-break-condition 1 1
^done
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",cond="1",
times="0",ignore="3"}}
(gdb)
```

Команда `-break-delete`

Краткое описание

`-break-delete (` *точка-останова* `)+`

Удалить точки останова, чьи номера указаны в списке аргументов. Это, очевидно, отражается на списке точек останова.

Команда GDB

Соответствующей командой GDB является ‘`delete`’.

Пример

```
(gdb)
-break-delete 1
^done
(gdb)
-break-list
^done,BreakpointTable={}
(gdb)
```

Команда `-break-disable`

Краткое описание

`-break-disable` (*точка-останова*)+

Отключить упомянутые *точки-останова*. Для указанных *точек-останова* поле ‘`enabled`’ в списке точек останова теперь установлено в ‘`n`’.

Команда GDB

Соответствующей командой GDB является ‘`disable`’.

Пример

```
(gdb)
-break-disable 2
^done
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="2",type="breakpoint",disp="keep",enabled="n",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0"}}
(gdb)
```

Команда `-break-enable`

Краткое описание

`-break-enable` (*точка-останова*)+

Включить (ранее отключенные) *точки-останова*.

Команда GDB

Соответствующей командой GDB является ‘`enable`’.

Пример

```
(gdb)
-break-enable 2
^done
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="2",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0"}}
(gdb)
```

Команда `-break-info`

Краткое описание

`-break-info` *точка-останова*

Получить информацию об одной точке останова.

Команда GDB

Соответствующей командой GDB является '`info break` *точка-останова*'.

Пример

Н.П.

Команда `-break-insert`

Краткое описание

`-break-insert [ -t ] [ -h ] [ -r ]`
`[ -c` *условие* `] [ -i` *счетчик-игнорирований* `]`
`[ -p` *нить* `] [` *строка* `|` *адрес* `]`

Строка, если указана, может быть одной из:

- функция
- имя-файла:номер-строки
- имя-файла:функция
- *адрес

Вот возможные необязательные параметры этой команды:

'`-t`' Вставить временную точку останова.

'`-h`' Вставить аппаратную точку останова.

'`-c` *условие*'
Сделать точку останова условной с заданным *условием*.

'`-i` *счетчик-игнорирований*'
Инициализировать *счетчик-игнорирований*.

'`-r`' Вставить обычную точку останова во всех функциях, чьи имена удовлетворяют данному регулярному выражению. Другие флаги к регулярному выражению неприменимы.

Результат

Результат имеет форму:

```
^done,bkptno="номер",func="имя-функции",
 file="имя-файла",line="ном-строки"
```

где *номер* является номером этой точки останова в GDB, *имя-функции*—имя функции, в которой была вставлена точка останова, *имя-файла*—имя исходного файла, в котором находится эта функция, а *ном-строки* является номером строки исходного текста в этом файле.

Замечание: этот формат может изменяться.

Команда GDB

Соответстующими командами GDB являются ‘`break`’, ‘`tbreak`’, ‘`hbreak`’, ‘`thbreak`’ и ‘`rbreak`’.

Пример

```
(gdb)
-break-insert main
^done,bkpt={number="1",addr="0x0001072c",file="recursive2.c",line="4"}
(gdb)
-break-insert -t foo
^done,bkpt={number="2",addr="0x00010774",file="recursive2.c",line="11"}
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x0001072c", func="main",file="recursive2.c",line="4",times="0"},
bkpt={number="2",type="breakpoint",disp="del",enabled="y",
addr="0x00010774",func="foo",file="recursive2.c",line="11",times="0"}}
(gdb)
-break-insert -r foo.*
~int foo(int, int);
^done,bkpt={number="3",addr="0x00010774",file="recursive2.c",line="11"}
(gdb)
```

Команда `-break-list`

Краткое описание

```
-break-list
```

Отображает список установленных точек останова, показывая следующие поля:

‘`Number`’ номер точки останова

‘`Type`’ тип точки останова: ‘`breakpoint`’ или ‘`watchpoint`’

‘`Disposition`’
эта точка останова должна быть удалена или отключена при срабатывании: ‘`keep`’ или ‘`nokeep`’

‘`Enabled`’ включена точка останова или нет: ‘`y`’ или ‘`n`’

‘Address’ местоположение в памяти, где установлена точка останова

‘What’ логическое положение точки останова, выраженное именем функции, именем файла, номером строки

‘Times’ Число раз, которое точка останова срабатывала

Если точек останова или наблюдения нет, поле BreakpointTable является пустым списком.

Команда GDB

Соответствующей командой GDB является ‘info break’.

Пример

```
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0"},
bkpt={number="2",type="breakpoint",disp="keep",enabled="y",
addr="0x00010114",func="foo",file="hello.c",line="13",times="0"}}
(gdb)
```

Вот пример результата, когда точек останова нет:

```
(gdb)
-break-list
^done,BreakpointTable={}
(gdb)
```

Команда -break-watch

Краткое описание

```
-break-watch [ -a | -r ]
```

Создать точку наблюдения. С ключем ‘-a’ будет создана точка наблюдения за *доступом*, то есть такая точка наблюдения, которая срабатывает либо при чтении, либо при записи в определенное местоположение в памяти. С ключем ‘-r’ созданная точка наблюдения будет точкой наблюдения за *чтением*, то есть она будет срабатывать только когда к определенному местоположению в памяти осуществляется доступ на чтение. Без этих ключей будет создана обычная точка наблюдения, то есть она будет срабатывать, когда к местоположению в памяти осуществляется доступ для записи. См. Раздел 5.1.2 [Установка точек наблюдения], с. 35.

Заметьте, что ‘-break-list’ выдаст единый список установленных точек наблюдения и останова.

Комада GDB

Соответствующими командами GDB являются ‘watch’, ‘awatch’ и ‘rwatch’.

Пример

Установка точки наблюдения за переменной в функции `main`:

```
(gdb)
-break-watch x
^done,wpt={number="2",exp="x"}
(gdb)
-exec-continue
^running
^done,reason="watchpoint-trigger",wpt={number="2",exp="x"},
value={old="-268439212",new="55"},
frame={func="main",args={},file="recursive2.c",line="5"}
(gdb)
```

Установка точки наблюдения за локальной переменной функции. GDB дважды остановит выполнение программы: сначала при изменении значения переменной, затем при выходе точки наблюдения из области видимости.

```
(gdb)
-break-watch C
^done,wpt={number="5",exp="C"}
(gdb)
-exec-continue
^running
^done,reason="watchpoint-trigger",
wpt={number="5",exp="C"},value={old="-276895068",new="3"},
frame={func="callee4",args={},
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="13"}
(gdb)
-exec-continue
^running
^done,reason="watchpoint-scope",wpnum="5",
frame={func="callee3",args={{name="strarg",
value="0x11940 \"A string argument.\""}},
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="18"}
(gdb)
```

Список точек останова и наблюдения, в разных местах выполнения программы. Заметьте, что как только точка наблюдения выходит из области видимости, она удаляется.

```
(gdb)
-break-watch C
^done,wpt={number="2",exp="C"}
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="8",times="1"},
bkpt={number="2",type="watchpoint",disp="keep",
enabled="y",addr="",what="C",times="0"}}
(gdb)
-exec-continue
^running
^done,reason="watchpoint-trigger",wpt={number="2",exp="C"},
```

```
value={old="-276895068",new="3"},
frame={func="callee4",args={},
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="13"}
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="8",times="1"},
bkpt={number="2",type="watchpoint",disp="keep",
enabled="y",addr="",what="C",times="-5"}}
(gdb)
-exec-continue
^running
^done,reason="watchpoint-scope",wpnum="2",
frame={func="callee3",args={{name="strarg",
value="0x11940 \"A string argument.\""}},
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="18"}
(gdb)
-break-list
^done,BreakpointTable={hdr={"Num","Type","Disp","Enb","Address","What"},
bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="8",times="1"}}
(gdb)
```

19.6 Управление данными GDB/MI

Этот раздел описывает команды GDB/MI для управления данными: исследование памяти и регистров, вычисление выражений, и так далее.

Команда `-data-disassemble`

Краткое описание

```
-data-disassemble
   [ -s нач-адр -e кон-адр ]
 | [ -f имя-файла -l ном-строки [ -n ст ] ]
 - режим
```

Где:

‘*нач-адр*’ начальный адрес (или `$pc`)

‘*кон-адр*’ конечный адрес

‘*имя-файла*’
имя файла для дисассемблирования

‘*ном-строки*’
номер строки, в районе которой проводить дисассемблирование

‘*ст*’ число строк дисассемблирования, которое необходимо произвести. Если равно -1 и *кон-адр* не указан, то будет дисассемблирована целая функция. Если

кон-адр указан и не равен нулю, и *ст* меньше, чем количество строк дисассемблирования между *нач-адр* и *кон-адр*, отображаются только *ст* строк. Если *ст* больше, чем число строк между *нач-адр* и *кон-адр*, отображаются только строки до *кон-адр*.

'*режим*' либо 0 (означает только результат дисассемблирования), либо 1 (означает смесь исходного текста и результата дисассемблирования).

Результат

Вывод для каждой инструкции состоит из четырех полей:

- Адрес
- Имя-Функции
- Смещение
- Инструкция

Заметьте, что то, что включено в поле инструкции, не обрабатывается непосредственно GDB/MI, то есть изменить его формат невозможно.

Команда GDB

Непосредственного отображения этой команды в CLI нет.

Пример

Дисассемблирование от текущего значения `$pc` до `$pc + 20`:

```
(gdb)
-data-disassemble -s $pc -e "$pc + 20" - 0
^done,
asm_insns={
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"},
{address="0x000107c8",func-name="main",offset="12",
inst="or  %o2, 0x140, %o1\t! 0x11940 <_lib_version+8>"},
{address="0x000107cc",func-name="main",offset="16",
inst="sethi  %hi(0x11800), %o2"},
{address="0x000107d0",func-name="main",offset="20",
inst="or  %o2, 0x168, %o4\t! 0x11968 <_lib_version+48>"}}
(gdb)
```

Дисассемблирование всей функции `main`. Строка 32 является частью `main`.

```
-data-disassemble -f basics.c -l 32 - 0
^done,asm_insns={
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"},
{address="0x000107c0",func-name="main",offset="4",
inst="mov   2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi %hi(0x11800), %o2"},
[...]
{address="0x0001081c",func-name="main",offset="96",inst="ret "},
```

```
{address="0x00010820",func-name="main",offset="100",inst="restore "}}
(gdb)
```

Дисассемблирование 3 инструкций от начала `main`:

```
(gdb)
-data-disassemble -f basics.c -l 32 -n 3 - 0
^done,asm_insns={
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"},
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"}}
(gdb)
```

Дисассемблирование 3 инструкций от начала `main` в смешаном режиме:

```
(gdb)
-data-disassemble -f basics.c -l 32 -n 3 - 1
^done,asm_insns={
src_and_asm_line={line="31",
file="/kwikemart/marge/ezannoni/flathead-dev/devo/gdb/ \
  testsuite/gdb.mi/basics.c",line_asm_insn={
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"}}},

src_and_asm_line={line="32",
file="/kwikemart/marge/ezannoni/flathead-dev/devo/gdb/ \
  testsuite/gdb.mi/basics.c",line_asm_insn={
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"}}}}
(gdb)
```

Команда `-data-evaluate-expression`

Краткое описание

`-data-evaluate-expression` *выраж*

Вычислить выражение *выраж*. Выражение может содержать подчиненный вызов функции. Вызов функции быдет выполнен синхронно. Если выражение содержит пробелы, оно должно быть заключено в двойные кавычки.

Команда GDB

Соответствующими командами GDB являются ‘`print`’, ‘`output`’ и ‘`call`’. В `gdbtk` есть соответствующая команда ‘`gdb_eval`’.

Пример

В следующем примере числа, предшествующие командам, суть *лексемы*. Для их описания, см. Раздел 19.1 [Синтаксис команд GDB/MI], с. 171. Обратите внимание на то, как GDB/MI возвращает те же лексемы в своем выводе.

```
211-data-evaluate-expression A
211^done,value="1"
(gdb)
311-data-evaluate-expression &A
311^done,value="0xefffeb7c"
(gdb)
411-data-evaluate-expression A+3
411^done,value="4"
(gdb)
511-data-evaluate-expression "A + 3"
511^done,value="4"
(gdb)
```

Команда `-data-list-changed-registers`

Краткое описание

```
-data-list-changed-registers
```

Выводит список регистров, которые изменились.

Команда GDB

GDB не имеет прямого аналога этой команды; соответствующей командой `gdbtk` является ‘`gdb_changed_register_list`’.

Пример

На плате PPC MBX:

```
(gdb)
-exec-continue
^running

(gdb)
*stopped,reason="breakpoint-hit",bkptno="1",frame={func="main",
args={},file="try.c",line="5"}
(gdb)
-data-list-changed-registers
^done,changed-registers={"0","1","2","4","5","6","7","8","9",
"10","11","13","14","15","16","17","18","19","20","21","22","23",
"24","25","26","27","28","30","31","64","65","66","67","69"}
(gdb)
```

Команда `-data-list-register-names`

Краткое описание

```
-data-list-register-names [ ( ном-рег )+ ]
```

Показать список имен регистров текущей цели. Если аргументы не заданы, показывается список имен всех регистров. Если в качестве аргументов заданы целые числа, команда напечатает список имен регистров, соответствующих аргументам.

Команда GDB

В GDB нет команды, которая соответствует ‘`-data-list-register-names`’. В `gdbtk` соответствующей командой является ‘`gdb_regnames`’.

Пример

Для платы PPC MBX:

```
(gdb)
-data-list-register-names
^done,register-names={"r0","r1","r2","r3","r4","r5","r6","r7",
"r8","r9","r10","r11","r12","r13","r14","r15","r16","r17","r18",
"r19","r20","r21","r22","r23","r24","r25","r26","r27","r28","r29",
"r30","r31","f0","f1","f2","f3","f4","f5","f6","f7","f8","f9",
"f10","f11","f12","f13","f14","f15","f16","f17","f18","f19","f20",
"f21","f22","f23","f24","f25","f26","f27","f28","f29","f30","f31",
"pc","ps","cr","lr","ctr","xer"}
(gdb)
-data-list-register-names 1 2 3
^done,register-names={"r1","r2","r3"}
(gdb)
```

Команда `-data-list-register-values`

Краткое описание

`-data-list-register-values` *фмт* `[ (` *ном-рег* `)*]`

Отобразить содержимое регистров. *Фмт* является форматом, в соответствии с которым должно быть возвращено содержимое регистров, за которым следует необязательный список чисел, указывающих регистры, подлежащие отображению. Отсутствие списка чисел означает, что должно быть возвращено содержимое всех регистров.

Вот допустимые форматы для *фмт*:

`x` Шестнадцатеричный

`o` Восьмеричный

`t` Двоичный

`d` Десятичный

`r` Без преобразования

`N` Натуральный

Команда GDB

Соответствующими командами GDB являются ‘`info reg`’, ‘`info all-reg`’ и (в `gdbtk`) ‘`gdb_fetch_registers`’.

Пример

Для платы PPC MBX (имейте ввиду: переносы строк даны только для удобства чтения, они не появляются в реальном выводе):

```
(gdb)
-data-list-register-values r 64 65
^done,register-values={{number="64",value="0xfe00a300"},
{number="65",value="0x00029002"}}
(gdb)
-data-list-register-values x
^done,register-values={{number="0",value="0xfe0043c8"},
{number="1",value="0x3fff88"},{number="2",value="0xfffffffe"},
{number="3",value="0x0"},{number="4",value="0xa"},
{number="5",value="0x3fff68"},{number="6",value="0x3fff58"},
{number="7",value="0xfe011e98"},{number="8",value="0x2"},
{number="9",value="0xfa202820"},{number="10",value="0xfa202808"},
{number="11",value="0x1"},{number="12",value="0x0"},
{number="13",value="0x4544"},{number="14",value="0xffdfffff"},
{number="15",value="0xffffffff"},{number="16",value="0xfffffeff"},
{number="17",value="0xefffffed"},{number="18",value="0xfffffffe"},
{number="19",value="0xffffffff"},{number="20",value="0xffffffff"},
{number="21",value="0xffffffff"},{number="22",value="0xfffffff7"},
{number="23",value="0xffffffff"},{number="24",value="0xffffffff"},
{number="25",value="0xffffffff"},{number="26",value="0xfffffffb"},
{number="27",value="0xffffffff"},{number="28",value="0xf7bfffff"},
{number="29",value="0x0"},{number="30",value="0xfe010000"},
{number="31",value="0x0"},{number="32",value="0x0"},
{number="33",value="0x0"},{number="34",value="0x0"},
{number="35",value="0x0"},{number="36",value="0x0"},
{number="37",value="0x0"},{number="38",value="0x0"},
{number="39",value="0x0"},{number="40",value="0x0"},
{number="41",value="0x0"},{number="42",value="0x0"},
{number="43",value="0x0"},{number="44",value="0x0"},
{number="45",value="0x0"},{number="46",value="0x0"},
{number="47",value="0x0"},{number="48",value="0x0"},
{number="49",value="0x0"},{number="50",value="0x0"},
{number="51",value="0x0"},{number="52",value="0x0"},
{number="53",value="0x0"},{number="54",value="0x0"},
{number="55",value="0x0"},{number="56",value="0x0"},
{number="57",value="0x0"},{number="58",value="0x0"},
{number="59",value="0x0"},{number="60",value="0x0"},
{number="61",value="0x0"},{number="62",value="0x0"},
{number="63",value="0x0"},{number="64",value="0xfe00a300"},
{number="65",value="0x29002"},{number="66",value="0x202f04b5"},
{number="67",value="0xfe0043b0"},{number="68",value="0xfe00b3e4"},
{number="69",value="0x20002b03"}}
(gdb)
```

Команда `-data-read-memory`

Краткое описание

```
-data-read-memory [ -o смещение ]
  адрес формат-слова размер-слова
  число-строк число-колонок [ ascимв ]
```

где:

‘*адрес*’ Выражение, определяющее адрес в памяти первого слова, которое надо прочитать. Сложные выражения, содержащие пробельные символы, должны заключаться в кавычки с использованием соглашений Си.

‘*формат-слова*’
Формат, который должен быть использован для печати слов памяти. Обозначения те же, что и для команды GDB `print` (см. Раздел 8.4 [Форматы вывода], с. 64).

‘*размер-слова*’
Размер в байтах каждого слова в памяти.

‘*число-строк*’
Число строк в выходной таблице.

‘*число-колонок*’
Число колонок в выходной таблице.

‘*ascимв*’ В настоящее время означает, что каждая строка должна включать ASCII-дамп. Значение *ascимв* используется в качестве заполняющего символа, когда байт не является элементом набора печатных знаков ASCII (печатные знаки ASCII это те знаки, чьи коды находятся между 32 и 126 включительно).

‘*смещение*’
Смещение, которое надо добавить к *адресу* перед тем, как начать извлечение из памяти.

Эта команда отображает содержимое памяти в виде таблицы из *число-строк* на *число-колонок* слов, причем каждое слово занимает *размер-слова* байт. В общей сложности считывается *число-строк* * *число-колонок* * *размер-слова* байт (возвращается как ‘`total-bytes`’). Если цель должна возвратить меньше запрошенного числа байт, отсутствующие слова идентифицируются при помощи ‘`N/A`’. Число байт, прочитанное с цели, возвращается в ‘`nr-bytes`’, а начальный адрес, использованный для чтении памяти, в ‘`addr`’.

Адрес следующей/предыдущей строки или страницы доступен в ‘`next-row`’ и ‘`prev-row`’, ‘`next-page`’ и ‘`prev-page`’.

Команда GDB

Соответствующей командой GDB является ‘`x`’. `gdbtk` имеет команду чтения памяти ‘`gdb_get_mem`’.

Пример

Прочитать шесть байт памяти, начиная с `bytes+6`, но сместиться на `-6` байт. Форматировать в три ряда по две колонки. Один байт на слово. Отображать каждое слово в шестнадцатеричном виде.

```
(gdb)
9-data-read-memory -o -6 - bytes+6 x 1 3 2
9^done,addr="0x00001390",nr-bytes="6",total-bytes="6",
```

```
next-row="0x00001396",prev-row="0x0000138e",next-page="0x00001396",
prev-page="0x0000138a",memory={
{addr="0x00001390",data={"0x00","0x01"}},
{addr="0x00001392",data={"0x02","0x03"}},
{addr="0x00001394",data={"0x04","0x05"}}}
(gdb)
```

Прочитать два байта памяти, начиная с адреса `shorts + 64` и отобразить в виде одного слова в десятичном виде.

```
(gdb)
5-data-read-memory shorts+64 d 2 1 1
5^done,addr="0x00001510",nr-bytes="2",total-bytes="2",
next-row="0x00001512",prev-row="0x0000150e",
next-page="0x00001512",prev-page="0x0000150e",memory={
{addr="0x00001510",data={"128"}}}
(gdb)
```

Прочитать тридцать два байта памяти, начиная с `bytes+16`, и форматировать на восемь рядов по четыре колонки. Включить строку, закодированную с использованием ‘`x`’ в качестве непечатного символа.

```
(gdb)
4-data-read-memory bytes+16 x 1 8 4 x
4^done,addr="0x000013a0",nr-bytes="32",total-bytes="32",
next-row="0x000013c0",prev-row="0x0000139c",
next-page="0x000013c0",prev-page="0x00001380",memory={
{addr="0x000013a0",data={"0x10","0x11","0x12","0x13"},ascii="xxxx"},
{addr="0x000013a4",data={"0x14","0x15","0x16","0x17"},ascii="xxxx"},
{addr="0x000013a8",data={"0x18","0x19","0x1a","0x1b"},ascii="xxxx"},
{addr="0x000013ac",data={"0x1c","0x1d","0x1e","0x1f"},ascii="xxxx"},
{addr="0x000013b0",data={"0x20","0x21","0x22","0x23"},ascii=" !\"#"},
{addr="0x000013b4",data={"0x24","0x25","0x26","0x27"},ascii="$%&'"},
{addr="0x000013b8",data={"0x28","0x29","0x2a","0x2b"},ascii="()*+"},
{addr="0x000013bc",data={"0x2c","0x2d","0x2e","0x2f"},ascii=",-./"}}
(gdb)
```

Команда `-display-delete`

Краткое описание

```
-display-delete число
```

Удалить элемент *номер* из списка выражений, подлежащих отображению.

Команда GDB

Соответствующей командой GDB является ‘`delete display`’.

Пример

Н.П.

Команда `-display-disable`

Краткое описание

`-display-disable` *номер*

Отключить элемент *номер* списка выражений, подлежащих отображению.

Команда GDB

Соответствующей командой GDB является ‘`disable display`’.

Пример

Н.П.

Команда `-display-enable`

Краткое описание

`-display-enable` *номер*

Включить элемент *номер* списка выражений, подлежащих отображению.

Команда GDB

Соответствующей командой GDB является ‘`enable display`’.

Пример

Н.П.

Команда `-display-insert`

Краткое описание

`-display-insert` *выражение*

Отображать *выражение* всякий раз, когда программа останавливается.

Команда GDB

Соответствующей командой GDB является ‘`display`’.

Пример

Н.П.

Команда `-display-list`

Краткое описание

`-display-list`

Перечислить элементы списка выражений, подлежащих автоматическому отображению. Текущие значения не показывать.

Команда GDB

Соответствующей командой GDB является ‘`info display`’.

Пример

Н.П.

Команда `-environment-cd`

Краткое описание

```
-environment-cd кат
```

Установить рабочий каталог GDB.

Команда GDB

Соответствующей командой GDB является ‘`cd`’.

Пример

```
(gdb)
-environment-cd /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done
(gdb)
```

Команда `-environment-directory`

Краткое описание

```
-environment-directory кат
```

Добавить каталог *кат* в начало пути поиска исходных файлов.

Команда GDB

Соответствующей командой GDB является ‘`dir`’.

Пример

```
(gdb)
-environment-directory /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done
(gdb)
```

Команда `-environment-path`

Краткое описание

```
-environment-path ( кат )+
```

Добавить каталоги *кат* в начало пути поиска объектных файлов.

Команда GDB

Соответствующей командой GDB является ‘path’.

Пример

```
(gdb)
-environment-path /kwikemart/marge/ezannoni/flathead-dev/ppc-eabi/gdb
^done
(gdb)
```

Команда -environment-pwd

Краткое описание

```
-environment-pwd
```

Показать текущий рабочий каталог.

Команда GDB

Соответствующей командой GDB является ‘pwd’.

Пример

```
(gdb)
-environment-pwd
~Working directory /kwikemart/marge/ezannoni/flathead-dev/devo/gdb.
^done
(gdb)
```

19.7 Управление программой GDB/MI

Завершение программы

В процессе выполнения, подчиненная программа может достигнуть конца, если она не встретит ни одной точки останова. В этом случае вывод будет включать код завершения, если программа завершилась ненормально.

Примеры

Программа завершилась нормально:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited-normally"
(gdb)
```

Программа завершилась ненормально:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited",exit-code="01"
(gdb)
```

Кроме того, программа может завершиться так, как если бы она получила сигнал, например `SIGINT`. В этом случае, GDB/MI отображает следующее:

```
(gdb)
*stopped,reason="exited-signalled",signal-name="SIGINT",
signal-meaning="Interrupt"
```

Команда `-exec-abort`

Краткое описание

```
-exec-abort
```

Убить выполняющуюся подчиненную программу.

Команда GDB

Соответствующей командой GDB является ‘`kill`’.

Пример

Н.П.

Команда `-exec-arguments`

Краткое описание

`-exec-arguments` *apr*

Установить аргументы подчиненной программы, которые должны быть использованы при следующем ‘`-exec-run`’.

Команда GDB

Соответствующей командой GDB является ‘`set args`’.

Пример

Пока нет.

Команда `-exec-continue`

Краткое описание

```
-exec-continue
```

Асинхронная команда. Возобновляет выполнение подчиненной программы до тех пор, пока не будет встречена точка останова, или пока подчиненная программа не завершится.

Команда GDB

Соответствующей командой GDB является ‘`continue`’.

Пример

```
-exec-continue
^running
(gdb)
@Hello world
*stopped,reason="breakpoint-hit",bkptno="2",frame={func="foo",args={},
file="hello.c",line="13"}
(gdb)
```

Команда `-exec-finish`

Краткое описание

```
-exec-finish
```

Асинхронная команда. Возобновляет выполнение подчиненной программы до тех пор, пока не завершится текущая функция. Отображает результаты, возвращенные функцией.

Команда GDB

Соответствующей командой GDB является ‘`finish`’.

Пример

Функция, возвращающая `void`.

```
-exec-finish
^running
(gdb)
@hello from foo
*stopped,reason="function-finished",frame={func="main",args={},
file="hello.c",line="7"}
(gdb)
```

Функция, возвращающая что-либо отличное от `void`. Печатается имя внутренней переменной GDB, хранящей результат, а также и сам результат.

```
-exec-finish
^running
(gdb)
*stopped,reason="function-finished",frame={addr="0x000107b0",func="foo",
args={{name="a",value="1"},{name="b",value="9"}},
file="recursive2.c",line="14"},
gdb-result-var="$1",return-value="0"
(gdb)
```

Команда `-exec-interrupt`

Краткое описание

```
-exec-interrupt
```

Асинхронная команда. Прерывает фоновое исполнение цели. Заметьте, что лексема, ассоциированная с сообщением об останове, совпадает с лексемой для выполнения команды, которая была прервана. Лексема для самого прерывания появляется только в выводе '`^done`'. Если пользователь пытается прервать невыполняющуюся программу, будет выведено сообщение об ошибке.

Команда GDB

Соответствующей командой GDB является '`interrupt`'.

Пример

```
(gdb)
111-exec-continue
111^running

(gdb)
222-exec-interrupt
222^done
(gdb)
111*stopped,signal-name="SIGINT",signal-meaning="Interrupt",
frame={addr="0x00010140",func="foo",args={},file="try.c",line="13"}
(gdb)

(gdb)
-exec-interrupt
^error,msg="mi_cmd_exec_interrupt: Inferior not executing."
(gdb)
```

Команда `-exec-next`

Краткое описание

```
-exec-next
```

Асинхронная команда. Возобновляет выполнение подчиненной программы, останавливая ее, когда достигается начало следующей строки исходного текста.

Команда GDB

Соответствующей командой GDB является '`next`'.

Пример

```
-exec-next
^running
(gdb)
*stopped,reason="end-stepping-range",line="8",file="hello.c"
(gdb)
```

Команда -exec-next-instruction

Краткое описание

```
-exec-next-instruction
```

Асинхронная команда. Выполняет одну машинную инструкцию. Если инструкция является вызовом функции, выполнение продолжается до возврата из функции. Если программа останавливается на инструкции в середине строки исходного текста, печатается также адрес.

Команда GDB

Соответствующей командой GDB является ‘`nexti`’.

Пример

```
(gdb)
-exec-next-instruction
^running

(gdb)
*stopped,reason="end-stepping-range",
addr="0x000100d4",line="5",file="hello.c"
(gdb)
```

Команда -exec-return

Краткое описание

```
-exec-return
```

Велит текущей функции немедленно вернуться. Не выполняет подчиненную программу. Отображает новый текущий кадр.

Команда GDB

Соответствующей командой GDB является ‘`return`’.

Пример

```
(gdb)
200-break-insert callee4
200^done,bkpt={number="1",addr="0x00010734",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="8"}
(gdb)
000-exec-run
000^running
(gdb)
000*stopped,reason="breakpoint-hit",bkptno="1",
frame={func="callee4",args={},
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="8"}
(gdb)
```

```
205-break-delete
205^done
(gdb)
111-exec-return
111^done,frame={level="0 ",func="callee3",
args={{name="strarg",
value="0x11940 \"A string argument.\""}},
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="18"}
(gdb)
```

Команда -exec-run

Краткое описание

```
-exec-run
```

Асинхронная команда. Начинает выполнение подчиненной программы с начала. Она выполняется до тех пор, пока либо не встретится точка останова, либо программа не завершится.

Команда GDB

Соответствующей командой GDB является ‘`run`’.

Пример

```
(gdb)
-break-insert main
^done,bkpt={number="1",addr="0x0001072c",file="recursive2.c",line="4"}
(gdb)
-exec-run
^running
(gdb)
*stopped,reason="breakpoint-hit",bkptno="1",
frame={func="main",args={},file="recursive2.c",line="4"}
(gdb)
```

Команда -exec-show-arguments

Краткое описание

```
-exec-show-arguments
```

Печатает аргументы программы.

Команда GDB

Соответствующей командой GDB является ‘`show args`’.

Пример

Н.П.

Команда -exec-step

Краткое описание

```
-exec-step
```

Асинхронная команда. Возобновляет выполнение подчиненной программы, останавливая ее, когда будет достигнуто начало следующей строки исходного файла, при условии, что она не является вызовом функции. Если же следующая строка является вызовом функции, программа останавливается на первой инструкции этой функции.

Команда GDB

Соответствующей командой GDB является ‘`step`’.

Пример

Пошаговый вход в функцию:

```
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",
frame={func="foo",args={{name="a",value="10"},
{name="b",value="0"}},file="recursive2.c",line="11"}
(gdb)
```

Обычное пошаговое выполнение:

```
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",line="14",file="recursive2.c"
(gdb)
```

Команда -exec-step-instruction

Краткое описание

```
-exec-step-instruction
```

Асинхронная команда. Возобновляет выполнение подчиненной программы, выполняя одну машинную инструкцию. Вывод, когда GDB остановится, будет различаться в зависимости от того, остановились мы в середине исходной строки или нет. В первом случае, адрес, по которому программа остановлена, также будет напечатан.

Команда GDB

Соответствующей командой GDB является ‘`stepi`’.

Пример

```
(gdb)
-exec-step-instruction
^running
```

```
(gdb)
*stopped,reason="end-stepping-range",
frame={func="foo",args={},file="try.c",line="10"}
(gdb)
-exec-step-instruction
^running

(gdb)
*stopped,reason="end-stepping-range",
frame={addr="0x000100f4",func="foo",args={},file="try.c",line="10"}
(gdb)
```

Команда `-exec-until`

Краткое описание

`-exec-until [` *местоположение* `]`

Асинхронная команда. Выполняет подчиненную программу до тех пор, пока не будет достигнуто указанное в аргументе *местоположение*. Если аргумента нет, подчиненная программа выполняется, пока не будет достигнута строка исходного текста, превышающая текущую. В этом случае, причиной остановки будет '`location-reached`'.

Команда GDB

Соответствующей командой GDB является '`until`'.

Пример

```
(gdb)
-exec-until recursive2.c:6
^running
(gdb)
x = 55
*stopped,reason="location-reached",frame={func="main",args={},
file="recursive2.c",line="6"}
(gdb)
```

Команда `-file-exec-and-symbols`

Краткое описание

`-file-exec-and-symbols` *файл*

Указать выполняемый файл для отладки. Это тот файл, из которого также читается таблица символов. Если файл не указан, команда очищает информацию о выполняемом файле и символах. Если при использовании этой команды без аргументов установлены точки останова, GDB выдаст сообщение об ошибке. В противном случае, никакого вывода не будет, за исключением уведомления о завершении.

Команда GDB

Соответствующей командой GDB является '`file`'.

Пример

```
(gdb)
-file-exec-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

Команда `-file-exec-file`

Краткое описание

`-file-exec-file` *файл*

Указать выполняемый файл для отладки. В отличие от '`-file-exec-and-symbols`', таблица символов *не* считывается из этого файла. При использовании без аргумента, GDB очищает информацию о выполняемом файле. Никакого вывода не производится, за исключением уведомления о завершении.

Команда GDB

Соответствующей командой GDB является '`exec-file`'.

Пример

```
(gdb)
-file-exec-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

Команда `-file-list-exec-sections`

Краткое описание

`-file-list-exec-sections`

Перечисляет разделы текущего выполняемого файла.

Команда GDB

Команда GDB '`info file`' показывает, помимо всего прочего, ту же информацию, что и эта команда. `gdbtk` имеет соответствующую команду '`gdb_load_info`'.

Пример

Н.П.

Команда `-file-list-exec-source-files`

Краткое описание

`-file-list-exec-source-files`

Перечисляет исходные файлы для текущего выполняемого файла.

Команда GDB

В GDB нет команды, непосредственно соответсвующей этой. `gdbtk` имеет аналогичную команду ‘`gdb_listfiles`’.

Пример

Н.П.

Команда `-file-list-shared-libraries`

Краткое описание

`-file-list-shared-libraries`

Перечисляет используемые программой разделяемые библиотеки.

Команда GDB

Соответствующей командой GDB является ‘`info shared`’.

Пример

Н.П.

Команда `-file-list-symbol-files`

Краткое описание

`-file-list-symbol-files`

Перечисляет файлы символов.

Команда GDB

Соответствующей командой GDB является ‘`info file`’ (ее часть).

Пример

Н.П.

Команда `-file-symbol-file`

Краткое описание

`-file-symbol-file` *файл*

Прочитать информацию символьной таблицы из указанного в аргументе *файла*. Будучи использованной без аргументов, очищает таблицу символьной информации GDB. Никакого вывода не производится, кроме уведомления о завершении.

Команда GDB

Соответствующей командой GDB является ‘`symbol-file`’.

Пример

```
(gdb)
-file-symbol-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

19.8 Разные команды GDB в GDB/MI

Команда -gdb-exit

Краткое описание

```
-gdb-exit
```

Немедленно выйти из GDB.

Команда GDB

Примерно соответствует команде ‘quit’.

Пример

```
(gdb)
-gdb-exit
```

Команда -gdb-set

Краткое описание

```
-gdb-set
```

Установить внутреннюю переменную GDB.

Команда GDB

Соответствующей командой GDB является ‘set’.

Пример

```
(gdb)
-gdb-set $foo=3
^done
(gdb)
```

Команда -gdb-show

Краткое описание

```
-gdb-show
```

Показать текущее значение переменной GDB.

Команда GDB

Соответствующей командой GDB является ‘`show`’.

Пример

```
(gdb)
-gdb-show annotate
^done,value="0"
(gdb)
```

Команда `-gdb-version`

Краткое описание

```
-gdb-version
```

Вывести информацию о версии GDB. Используется преимущественно при тестировании.

Команда GDB

Эквивалентной команды GDB нет. По умолчанию, GDB показывает эту информацию, когда вы вызываете интерактивный сеанс.

Пример

```
(gdb)
-gdb-version
~GNU gdb 5.2.1
~Copyright 2000 Free Software Foundation, Inc.
~GDB is free software, covered by the GNU General Public License, and
~you are welcome to change it and/or distribute copies of it under
~ certain conditions.
~Type "show copying" to see the conditions.
~There is absolutely no warranty for GDB.  Type "show warranty" for
~ details.
~This GDB was configured as
 "-host=sparc-sun-solaris2.5.1 -target=ppc-eabi".
^done
(gdb)
```

19.9 Команды управления стеком в GDB/MI

Команда `-stack-info-frame`

Краткое описание

```
-stack-info-frame
```

Получить информацию о текущем кадре.

Команда GDB

Соответствующей командой GDB является ‘`info frame`’ или ‘`frame`’ (без аргументов).

Пример

Н.П.

Команда `-stack-info-depth`

Краткое описание

```
-stack-info-depth [ макс-глуб ]
```

Возвращает глубину стека. Если указан целочисленный аргумент *макс-глуб*, не считать более *макс-глуб* кадров.

Команда GDB

Эквивалентной команды GDB нет.

Пример

Для стека с уровнями кадров от 0 до 11:

```
(gdb)
-stack-info-depth
^done,depth="12"
(gdb)
-stack-info-depth 4
^done,depth="4"
(gdb)
-stack-info-depth 12
^done,depth="12"
(gdb)
-stack-info-depth 11
^done,depth="11"
(gdb)
-stack-info-depth 13
^done,depth="12"
(gdb)
```

Команда `-stack-list-arguments`

Краткое описание

```
-stack-list-arguments показ-знач
   [ нижн-кадр верхн-кадр ]
```

Отобразить список аргументов для кадров от *нижн-кадр* до *верхн-кадр* (включительно). Если *нижн-кадр* и *верхн-кадр* не указаны, перечислить аргументы для всего стека вызовов.

Аргумент *показ-знач* должен иметь значение 0 или 1. Значение 0 означает, что выводятся только имена аргументов, а 1,—что печатаются как имена, так и значения аргументов.

Команда GDB

GDB не имеет эквивалентной команды. В `gdbtk` есть команда '`gdb_get_args`', которая частично перекрывается с действием '`-stack-list-arguments`'.

Пример

```
(gdb)
-stack-list-frames
^done,
stack={
frame={level="0 ",addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="8"},
frame={level="1 ",addr="0x0001076c",func="callee3",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="17"},
frame={level="2 ",addr="0x0001078c",func="callee2",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="22"},
frame={level="3 ",addr="0x000107b4",func="callee1",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="27"},
frame={level="4 ",addr="0x000107e0",func="main",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="32"}}
(gdb)
-stack-list-arguments 0
^done,
stack-args={
frame={level="0",args={}},
frame={level="1",args={name="strarg"}},
frame={level="2",args={name="intarg",name="strarg"}},
frame={level="3",args={name="intarg",name="strarg",name="fltarg"}},
frame={level="4",args={}}}
(gdb)
-stack-list-arguments 1
^done,
stack-args={
frame={level="0",args={}},
frame={level="1",
 args={{name="strarg",value="0x11940 \"Строковый аргумент.\""}}},
frame={level="2",args={
{name="intarg",value="2"},
{name="strarg",value="0x11940 \"Строковый аргумент.\""}}},
{frame={level="3",args={
{name="intarg",value="2"},
{name="strarg",value="0x11940 \"Строковый аргумент.\""},
{name="fltarg",value="3.5"}}},
frame={level="4",args={}}}
(gdb)
-stack-list-arguments 0 2 2
^done,stack-args={frame={level="2",args={name="intarg",name="strarg"}}}
(gdb)
-stack-list-arguments 1 2 2
^done,stack-args={frame={level="2",
args={{name="intarg",value="2"},
```

```
{name="strarg",value="0x11940 \"Строковый аргумент.\""}}}}
(gdb)
```

Команда -stack-list-frames

Краткое описание

```
-stack-list-frames [ нижн-кадр верхн-кадр ]
```

Перечисляет кадры, находящиеся в данный момент в стеке. Для каждого кадра, команда отображает следующую информацию:

‘*level*’ Номер кадра, 0 для самого верхнего, то есть для самой внутренней функции.

‘*addr*’ Значение `$pc` для этого кадра.

‘*func*’ Имя функции.

‘*file*’ Имя исходного файла, где находится функция.

‘*line*’ Номер строки, соответствующий `$pc`.

Будучи вызванной без аргументов, эта команда печатает цепочку вызовов для всего стека. Если задано два целочисленных аргумента, она показывает кадры с уровнями между этими аргументами (включительно). Если аргументы равны, она показывает один единственный кадр соответствующего уровня.

Команда GDB

Соответствующими командами GDB являются ‘`backtrace`’ и ‘`where`’.

Пример

Цепочка вызовов стека целиком:

```
(gdb)
-stack-list-frames
^done,stack=
{frame={level="0 ",addr="0x0001076c",func="foo",
  file="recursive2.c",line="11"},
frame={level="1 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="2 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="3 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="4 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="5 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="6 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="7 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="8 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
```

```
frame={level="9 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="10",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="11",addr="0x00010738",func="main",
  file="recursive2.c",line="4"}}
(gdb)
```

Показать кадры между *нижн-кадр* и *верхн-кадр*:

```
(gdb)
-stack-list-frames 3 5
^done,stack=
{frame={level="3 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="4 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="5 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"}}
(gdb)
```

Показать один кадр:

```
(gdb)
-stack-list-frames 3 3
^done,stack=
{frame={level="3 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"}}
(gdb)
```

Команда `-stack-list-locals`

Краткое описание

`-stack-list-locals` *печатать-значения*

Вывести имена локальных переменных для текущего кадра. С параметром 0 выводит только имена переменных, с параметром 1 выводит также их значения.

Команда GDB

‘`info locals`’ в GDB, ‘`gdb_get_locals`’ в `gdbtk`.

Пример

```
(gdb)
-stack-list-locals 0
^done,locals={name="A",name="B",name="C"}
(gdb)
-stack-list-locals 1
^done,locals={{name="A",value="1"},{name="B",value="2"},
  {name="C",value="3"}}
(gdb)
```

Команда `-stack-select-frame`

Краткое описание

`-stack-select-frame` *ном-кадра*

Изменить текущий кадр. Выбрать другой кадр *ном-кадра* в стеке.

Команда GDB

Соответствующими командами GDB являются ‘`frame`’, ‘`up`’, ‘`down`’, ‘`select-frame`’, ‘`up-silent`’ и ‘`down-silent`’.

Пример

```
(gdb)
-stack-select-frame 2
^done
(gdb)
```

19.10 Команды GDB/MI запросов о символах

Команда `-symbol-info-address`

Краткое описание

`-symbol-info-address` *символ*

Описать, где хранится *символ*.

Команда GDB

Соответствующей командой GDB является ‘`info address`’.

Пример

Н.П.

Команда `-symbol-info-file`

Краткое описание

`-symbol-info-file`

Показать файл для символа.

Команда GDB

Эквивалентной команды GDB нет. В `gdbtk` есть команда ‘`gdb_find_file`’.

Пример

Н.П.

Команда `-symbol-info-function`

Краткое описание

`-symbol-info-function`

Показать, в какой функции находится символ.

Команда GDB

'`gdb_get_function`' в `gdbtk`.

Пример

Н.П.

Команда `-symbol-info-line`

Краткое описание

`-symbol-info-line`

Показать адреса памяти кода для текущей строки.

Команда GDB

Соответствующей командой GDB является '`info line`'. В `gdbtk` есть команды '`gdb_get_line`' и '`gdb_get_file`'.

Пример

Н.П.

Команда `-symbol-info-symbol`

Краткое описание

`-symbol-info-symbol` *адрес*

Описать, какой символ находится в местоположении *адрес*.

Команда GDB

Соответствующей командой GDB является '`info symbol`'.

Пример

Н.П.

Команда `-symbol-list-functions`

Краткое описание

```
-symbol-list-functions
```

Перечислить функции, находящиеся в выполняемом файле.

Команда GDB

‘`info functions`’ в GDB, ‘`gdb_listfunc`’ и ‘`gdb_search`’ в `gdbtk`.

Пример

Н.П.

Команда `-symbol-list-types`

Краткое описание

```
-symbol-list-types
```

Перечислить все имена типов.

Команда GDB

Соответствующими командами являются ‘`info types`’ в GDB и ‘`gdb_search`’ в `gdbtk`.

Пример

Н.П.

Команда `-symbol-list-variables`

Краткое описание

```
-symbol-list-variables
```

Перечислить имена всех глобальных и статических переменных.

Команда GDB

‘`info variables`’ в GDB, ‘`gdb_search`’ в `gdbtk`.

Пример

Н.П.

Команда `-symbol-locate`

Краткое описание

```
-symbol-locate
```

Команда GDB

‘gdb_loc’ в gdbtk.

Пример

Н.П.

Команда -symbol-type

Краткое описание

`-symbol-type` *переменная*

Показать тип *переменной*.

Команда GDB

Соответствующей командой GDB является ‘ptype’, в gdbtk есть команда ‘gdb_obj_variable’.

Пример

Н.П.

19.11 Команды GDB/MI управления целью

Команда -target-attach

Краткое описание

`-target-attach` *ид-проц* | *файл*

Присоединиться к процессу *ид-проц* или файлу *файл* вне GDB.

Команда GDB

Соответствующей командой GDB является ‘attach’.

Пример

Н.П.

Команда -target-compare-sections

Краткое описание

`-target-compare-sections [` *раздел* `]`

Сравнить данные раздела *раздел* на цели с выполняемым файлом. Без аргумента сравниваются все разделы.

Команда GDB

Эквивалентной командой в GDB является ‘`compare-sections`’.

Пример

Н.П.

Команда `-target-detach`

Краткое описание

```
-target-detach
```

Отсоединиться от удаленной цели. Никакого вывода не производится.

Команда GDB

Соответствующей командой GDB является ‘`detach`’.

Пример

```
(gdb)
-target-detach
^done
(gdb)
```

Команда `-target-download`

Краткое описание

```
-target-download
```

Загружает выполняемый файл на удаленную цель. Команда два раза в секунду обновляет сообщение, которое включает поля:

‘`section`’ Имя раздела.

‘`section-sent`’
Размер переданного на данный момент участка для этого раздела.

‘`section-size`’
Размер раздела.

‘`total-sent`’
Общий размер переданной на данный момент информации (текущий и предыдущие разделы).

‘`total-size`’
Размер всего выполняемого файла, подлежащего загрузке.

Каждое сообщение посылается в виде сообщения статуса (см. Раздел 19.1.2 [Выходной синтаксис GDB/MI], с. 172).

Кроме того, команда печатает имена и размеры разделов, по мере их загрузки. Эти сообщения включают следующие поля:

‘`section`’ Имя раздела.

‘section-size’
: Размер раздела.

‘total-size’
: Размер всего выполняемого файла, подлежащего загрузке.

В конце печатается суммарная информация.

Команда GDB

Соответствующей командой GDB является ‘load’.

Пример

Замечание: каждое сообщение статуса появляется на одной строке. Здесь сообщения были разбиты на несколько строк, чтобы они могли влезть на страницу.

```
(gdb)
-target-download
+download,{section=".text",section-size="6668",total-size="9880"}
+download,{section=".text",section-sent="512",section-size="6668",
total-sent="512",total-size="9880"}
+download,{section=".text",section-sent="1024",section-size="6668",
total-sent="1024",total-size="9880"}
+download,{section=".text",section-sent="1536",section-size="6668",
total-sent="1536",total-size="9880"}
+download,{section=".text",section-sent="2048",section-size="6668",
total-sent="2048",total-size="9880"}
+download,{section=".text",section-sent="2560",section-size="6668",
total-sent="2560",total-size="9880"}
+download,{section=".text",section-sent="3072",section-size="6668",
total-sent="3072",total-size="9880"}
+download,{section=".text",section-sent="3584",section-size="6668",
total-sent="3584",total-size="9880"}
+download,{section=".text",section-sent="4096",section-size="6668",
total-sent="4096",total-size="9880"}
+download,{section=".text",section-sent="4608",section-size="6668",
total-sent="4608",total-size="9880"}
+download,{section=".text",section-sent="5120",section-size="6668",
total-sent="5120",total-size="9880"}
+download,{section=".text",section-sent="5632",section-size="6668",
total-sent="5632",total-size="9880"}
+download,{section=".text",section-sent="6144",section-size="6668",
total-sent="6144",total-size="9880"}
+download,{section=".text",section-sent="6656",section-size="6668",
total-sent="6656",total-size="9880"}
+download,{section=".init",section-size="28",total-size="9880"}
+download,{section=".fini",section-size="28",total-size="9880"}
+download,{section=".data",section-size="3156",total-size="9880"}
+download,{section=".data",section-sent="512",section-size="3156",
total-sent="7236",total-size="9880"}
+download,{section=".data",section-sent="1024",section-size="3156",
total-sent="7748",total-size="9880"}
+download,{section=".data",section-sent="1536",section-size="3156",
```

```
total-sent="8260",total-size="9880"}
+download,{section=".data",section-sent="2048",section-size="3156",
total-sent="8772",total-size="9880"}
+download,{section=".data",section-sent="2560",section-size="3156",
total-sent="9284",total-size="9880"}
+download,{section=".data",section-sent="3072",section-size="3156",
total-sent="9796",total-size="9880"}
^done,address="0x10004",load-size="9880",transfer-rate="6586",
write-rate="429"
(gdb)
```

Команда `-target-exec-status`

Краткое описание

```
-target-exec-status
```

Предоставить информацию о состоянии цели (например, выполняется она или нет).

Команда GDB

Эквивалентной команды GDB нет.

Пример

Н.П.

Команда `-target-list-available-targets`

Краткое описание

```
-target-list-available-targets
```

Перечислить цели, к которым можно установить соединение.

Команда GDB

Соответствующей командой GDB является ‘`help target`’.

Пример

Н.П.

Команда `-target-list-current-targets`

Краткое описание

```
-target-list-current-targets
```

Описать текущую цель.

Команда GDB

Соответствующая информация (вместе с другой) печатается командой ‘`info file`’.

Пример

Н.П.

Команда `-target-list-parameters`

Краткое описание

```
-target-list-parameters
```

Команда GDB

Эквивалента нет.

Пример

Н.П.

Команда `-target-select`

Краткое описание

`-target-select` *тип параметры ...*

Соединить GDB с удаленной целью. Эта команда допускает два аргумента:

‘*тип*’ Тип цели, например, ‘`async`’, ‘`remote`’, и т.д.

‘*параметры*’
Имена устройств, названия машин и тому подобное. См. Раздел 13.2 [Команды для управления целями], с. 111, для более полной информации.

Результатом является уведомление о соединении, за которым следует адрес, по которому находится целевая программа, в следующей форме:

```
^connected,addr="адрес",func="имя функции",
  args={список аргументов}
```

Команда GDB

Соответствующей командой GDB является ‘`target`’.

Пример

```
(gdb)
-target-select async /dev/ttya
^connected,addr="0xfe00a300",func="??",args={}
(gdb)
```

19.12 Команды GDB/MI для нитей

Команда `-thread-info`

Краткое описание

```
-thread-info
```

Команда GDB

Эквивалента нет.

Пример

Н.П.

Команда `-thread-list-all-threads`

Краткое описание

```
-thread-list-all-threads
```

Команда GDB

Эквивалентной командой GDB является ‘`info threads`’.

Пример

Н.П.

Команда `-thread-list-ids`

Краткое описание

```
-thread-list-ids
```

Выводит список известных GDB в данный момент идентификаторов нитей. В конце списка также выводится общее число таких нитей.

Команда GDB

Часть ‘`info threads`’ предоставляет ту же информацию.

Пример

Кроме основного процесса нет ни одной нити:

```
(gdb)
-thread-list-ids
^done,thread-ids={},number-of-threads="0"
(gdb)
```

Несколько нитей:

```
(gdb)
-thread-list-ids
^done,thread-ids={thread-id="3",thread-id="2",thread-id="1"},
number-of-threads="3"
(gdb)
```

Команда `-thread-select`

Краткое описание

`-thread-select` *номер-нити*

Сделать нить *номер-нити* текущей. Команда выводит номер новой текущей нити и самый верхний кадр для нее.

Команда GDB

Соответствующей командой GDB является ‘`thread`’.

Пример

```
(gdb)
-exec-next
^running
(gdb)
*stopped,reason="end-stepping-range",thread-id="2",line="187",
file="../../../devo/gdb/testsuite/gdb.threads/linux-dp.c"
(gdb)
-thread-list-ids
^done,
thread-ids={thread-id="3",thread-id="2",thread-id="1"},
number-of-threads="3"
(gdb)
-thread-select 3
^done,new-thread-id="3",
frame={level="0 ",func="vprintf",
args={{name="format",value="0x8048e9c \"%*s%c %d %c\\n\""},
{name="arg",value="0x2"}},file="vprintf.c",line="31"}
(gdb)
```

19.13 Команды GDB/MI для точек трассировки

Команды для точек трассировки еще не реализованы.

19.14 Изменяемые объекты GDB/MI

Обоснование для изменяемых объектов в GDB/MI

Для реализации изменяемого отладочного окна (локальные переменные, наблюдаемые выражения, и т.д.), мы предлагаем модификацию существующего кода, используемого в `Insight`.

Вот две основные причины для этого:

1. Он был проверен на практике (это уже его второе поколение).
2. Это сократит время разработки (не стоит говорить, как это сейчас важно).

Первоначальный интерейс был разработан для использование кодом Tcl, так что он был немного изменен, чтобы его можно было использовать через GDB/MI. Этот раздел описывает операции GDB/MI, которые будут доступны, и дает некоторые советы по их использованию.

Замечание: В дополнение к описанному здесь набору операций, мы ожидаем, что GUI-реализация изменяемого окна будет требовать, как минимум, следующие операции:

- `-gdb-show output-radix`
- `-stack-list-arguments`
- `-stack-list-locals`
- `-stack-select-frame`

Введение в изменяемые объекты в GDB/MI

Основной идеей изменяемых объектов является создание именованного объекта для представления переменной, выражения, местоположения в памяти или даже регистра ЦП. Для каждого созданного объекта существует набор операций для изучения или изменения его свойств.

Более того, сложные типы данных, такие как структуры Си, представлены в древовидном формате. Например, переменная типа `struct` является корнем, а потомки будут представлять элементы этой структуры. Если потомок сам является сложным типом, он также будет иметь своих потомков. Соответствующие различия языков учитываются для Си, Си++ и Java.

При возврате реальных значений объектов, эта возможность позволяет отдельно выбирать формат отображения, используемый при создании результата. Он может быть выбран из: двоичный, десятичный, шестнадцатеричный, восьмеричный и обычный. Обычный ссылается на формат по умолчанию, выбираемый автоматически в зависимости от типа переменной (например, десятичный для `int`, шестнадцатеричный для указателей, и т.д.).

Далее следует полный набор операций GDB/MI, определенный для доступа к этим возможностям:

Операция	**Описание**
`-var-create`	создать изменяемый объект
`-var-delete`	удалить изменяемый объект и его потомков
`-var-set-format`	установить формат отображения для этой переменной
`-var-show-format`	показать формат отображения для этой переменной
`-var-info-num-children`	сообщает, сколько потомков имеет данный объект
`-var-list-children`	возвращает список потомков объекта
`-var-info-type`	показать тип этого изменяемого объекта
`-var-info-expression`	напечатать, что представляет этот изменяемый объект
`-var-show-attributes`	является ли эта переменная редактируемой? она здесь существует?
`-var-evaluate-expression`	получить значение этой переменной
`-var-assign`	установить значение этой переменной
`-var-update`	скорректировать переменную и ее потомков

В следующем подразделе мы подробно описываем каждую операцию и предлагаем возможный способ ее использования.

Описание и использование операций для изменяемых объектов

Команда `-var-create`

Краткое описание

`-var-create {`*имя* `| "-"}`
`{`*адрес-кадра* `| "*"}` *выражение*

Данная операция создает изменяемый объект. Это позволяет наблюдать за переменной, результатом выражения, ячейкой памяти или регистром ЦП.

Параметр *имя* является строкой, по которой можно ссылаться на объект. Она должен быть уникальной. Если указан '`-`', система изменяемых объектов автоматически сгенерирует строку "varNNNNNN". Она будет уникальной, при условии, что *имя* не будет указано в этом формате. Команда завершается ошибкой, если найдено повторяющееся имя.

В *адресе-кадра* может быть задан кадр, в котором должно быть вычислено выражение. '`*`' указывает, что должен использоваться текущий кадр.

Выражение—это произвольное выражение, правильное в текущем наборе языков (не должно начинаться со '`*`'), или одно из следующего:

- '`*`*адрес*', где *адрес* есть адрес ячейки памяти
- '`*`*адрес*`-`*адрес*' — диапазон адресов памяти (TBD)
- '`$`*имя-рег*' — имя регистра ЦП

Результат

Эта операция возвращает имя, число потомков и тип созданного объекта. Тип возвращается как строка, как будто она создана GDB CLI:

`name="`*имя*`",numchild="N",type="`*тип*`"`

Команда `-var-delete`

Краткое описание

`-var-delete` *имя*

Удаляет созданный ранее изменяемый объект и всех его потомков.

Возвращает ошибку, если объект с именем *имя* не найден.

Команда `-var-set-format`

Краткое описание

`-var-set-format` *имя спецификатора-формата*

Устанавливает формат вывода в *спецификаторе-формата* для значения объекта *имя*.

Синтаксис *спецификатора-формата* следующий:

спецификатор-формата $\mapsto$
`{binary | decimal | hexadecimal | octal | natural}`

Команда `-var-show-format`

Краткое описание

`-var-show-format` *имя*

Возвращает формат, используемый для отображения значений объекта *имя*.

формат ↦
специф-формата

Команда `-var-info-num-children`

Краткое описание

`-var-info-num-children` *имя*

Возвращает число потомков изменяемого объекта с именем *имя*:

`numchild=`*n*

Команда `-var-list-children`

Краткое описание

`-var-list-children` *имя*

Возвращает список потомков указанного изменяемого объекта:

`numchild=`*n*`,children={{name=`*имя*`,`
`numchild=`*n*`,type=`*тип*`},`(повторяется N раз)`}`

Команда `-var-info-type`

Краткое описание

`-var-info-type` *имя*

Возвращает тип указанного изменяемого объекта *имя*. Тип возвращается как строка в том же формате, в котором она выдается GDB CLI:

`type=`*имя-типа*

Команда `-var-info-expression`

Краткое описание

`-var-info-expression` *имя*

Возвращает то, что представляет изменяемый объект с именем *имя*:

`lang=`*специф-языка*`,exp=`*выражение*

где *специф-языка* есть `{"C" | "C++" | "Java"}`.

Команда `-var-show-attributes`

Краткое описание

`-var-show-attributes` *имя*

Перечисляет атрибуты заданного изменяемого объекта *имя*:

`status=`*атр* `[ ( ,`*атр* `)* ]`

где *атр* есть `{ { editable | noneditable } | TBD }`.

Команда `-var-evaluate-expression`

Краткое описание

`-var-evaluate-expression` *имя*

Вычисляет выражение, которое представлено указанным изменяемым объектом и возвращает его значение в виде строки в текущем формате, определенном для объекта:

`value=`*значение*

Команда `-var-assign`

Краткое описание

`-var-assign` *имя выражение*

Присваивает значение *выражения* изменяемому объекту, заданному *именем*. Объект должен быть в состоянии ‘`editable`’.

Команда `-var-update`

Краткое описание

`-var-update {`*имя* `| "*"}`

Обновить значение изменяемого объекта с именем *имя* путем вычисления его выражения, после получения всех новых значений из памяти или регистров. ‘`*`’ приводит к обновлению всех существующих изменяемых объектов.

19.15 Черновик изменений к выходному синтаксису GDB/MI

Одной проблемой, обнаруженной в существующем синтаксисе вывода GDB/MI, была трудность в нахождении отличий между наборами вроде этого:

```
{number="1",type="breakpoint",disp="keep",enabled="y"}
```

где каждое значение имеет уникальную метку, и таким списком:

```
{"1","2","4"}
{bp="1",bp="2",bp="4"}
```

где значения не имеют меток, или метки повторяются.

Далее представлен черновик изменений к спецификации вывода, который решает эту проблему.

Вывод GDB/MI состоит из нуля или более внеочередных записей, за которыми может следовать одна результирующая запись, причем она относится к самой последней введенной команде. Последовательность завершается ‘`gdb`’.

Асинхронный вывод GDB/MI такой же.

Каждая выходная запись, непосредственно связанная с входной командой, начинается с префикса *лексема* входной команды.

вывод ↦ `{` *внеочередная-запись* `}` `[` *рез-запись* `]` `"(gdb)"` *nl*

рез-запись ↦
`[` *лексема* `]` `"^"` *рез-класс* `{` `","` *результат* `}` *nl*

внеочередная-запись ↦
асинхр-запись | *поточн-запись*

асинхр-запись ↦
асинхр-вывод-выполн | *асинхр-вывод-статуса* | *асинхр-вывод-уведомл*

асинхр-вывод-выполн ↦
`[` *лексема* `]` `"*"` *асинхр-вывод*

асинхр-вывод-статуса ↦
`[` *лексема* `]` `"+"` *асинхр-вывод*

асинхр-вывод-уведомл ↦
`[` *лексема* `]` `"="` *асинхр-вывод*

асинхр-вывод ↦
асинхр-класс `{` `","` *результат* `}` *nl*

результ-класс ↦
"done" | "running" | "connected" | "error" | "exit"

асинхр-класс ↦
"stopped" | *другое по необходимости, т.к. находится в стадии разработки*

результат ↦
строка `"="` *значение*

значение ↦
строка-си | *набор* | *список*

набор ↦ `"{}"` | `"{"` *результат* `{` `","` *результат* `}` `"}"`

список ↦ `"[]"` | `"["` *значение* `{` `","` *значение* `}` `"]"`

строка ↦ *[-A-Za-z\.0-9_]**

строка-си ↦
Смотри спецификацию ввода

поточн-запись ↦
консольн-поточн-вывод | *целев-поточн-вывод* | *журн-поточн-вывод*

консольн-поточн-вывод ↦
`"~"` *строка-си*

целев-поточн-вывод ↦
`"@"` *строка-си*

журн-поточн-вывод ↦
`"&"` *строка-си*

nl ↦ CR | CR-LF

лексема ↦
"любая последовательность цифр"

Кроме того, следующее находится в стадии разработки.

запрос Это действие пока не определено.

Замечания:

- Все выходные последовательности заканчиваются строкой, содержащей точку.
- *Лексема* берется из соответствующего запроса. Если выполняющаяся команда прерывается командой `-exec-interrupt`, лексема, ассоциированная с сообщением '*stopped', берется из исходной выполнявшейся команды, а не из команды прерывания.
- *Асинхр-вывод-статуса* содержит информацию о статусе выполнения медленной операции. Она может быть отброшена. Вся информация о статусе начинается с префикса '`+`'.
- *Асинхр-вывод-выполн* содержит изменения асинхронного состояния на цели (остановлена, запущена, исчезла). Весь асинхронный вывод начинается с префикса '`*`'.
- *Асинхр-вывод-уведомл* содержит дополнительную информацию, которую должен обработать клиент (информация о новой точке останова). Весь уведомительный вывод начинается с префикса '`=`'.
- *Консольн-поточн-вывод* является выводом, который должен быть отображен на консоли без изменений. Это текстовая реакция на команду CLI. Весь консольный вывод начинается с префикса '`~`'.
- *Целев-поточн-вывод* является выводом, произведенным целевой программой. Весь целевой вывод начинается с префикса '`@`'.
- *Журн-поточн-вывод* является текстом, исходящим изнутри GDB, например, сообщения, которые должны быть выведены как часть журнала ошибок. Весь журнальный вывод начинается с префикса '`&`'.

20 Отчеты об ошибках в GDB

Ваши отчеты об ошибках играют существенную роль в обеспечении надежности GDB.

Сообщение об ошибке может помочь вам найти решение вашей проблемы, а может и не помочь. Но в любом случае, основная функция отчета об ошибке—помочь всему обществу сделать следующую версию GDB лучше. Отчеты об ошибках—это ваш вклад в поддержку GDB.

Чтобы отчет об ошибке сделал свое дело, вы должны включить в него информацию, которая даст нам возможность ее устранить.

20.1 Вы нашли ошибку?

Если вы не уверены, нашли ли вы ошибку, вот несколько руководящих принципов:

- Если отладчик получает фатальный сигнал, то это ошибка в GDB, независимо от ввода. В надежных отладчиках сбоев не бывает.
- Если GDB выводит сообщение об ошибке для допустимого ввода—это ошибка. (Заметьте, что если вы выполняете кросс-отладку, проблема может возникать где-то в соединении к цели.)
- Если GDB не выводит сообщение об ошибке для недопустимого ввода, это ошибка. Однако вы должны обратить внимание, что если по вашему мнению что-то является “недопустимым вводом”, по нашему мнению это может быть “расширением” или “поддержкой для традиционной практики”.
- Если вы опытный пользователь средств отладки, ваши предложения по улучшению GDB приветствуются в любом случае.

20.2 Как составлять отчеты об ошибках

Некоторые компании и частные лица предлагают поддержку для программных продуктов GNU. Если вы получили GDB из организации поддержки, мы рекомендуем вам сперва связаться с ней.

Вы можете найти контактную информацию для многих организаций поддержки и частных лиц в файле ‘`etc/SERVICE`’ в дистрибутиве GNU Emacs.

В любом случае, мы также рекомендуем вам послать отчет об ошибке в GDB по этому адресу:

`bug-gdb@gnu.org`

Не посылайте отчеты об ошибках в ‘`info-gdb`’, или в ‘`help-gdb`’, или в какую-либо группу новостей. Большинство пользователей GDB не хотят получать отчеты об ошибках. Те, кто этого действительно хочет, должны получать ‘`bug-gdb`’.

Список рассылки ‘`bug-gdb`’ имеет группу новостей ‘`gnu.gdb.bug`’, которая служит как повторитель. Список рассылки и группа новостей имеют в точности одинаковые сообщения. Часто люди посылают сообщения об ошибках в группу новостей, вместо отправки по электронной почте. Это работает, но имеется одна проблема, которая может быть решающей: группа новостей часто не имеет информации об обратном адресе отправителя. Таким образом, если нам потребуется запросить дополнительную информацию, мы можем не иметь возможности связаться с вами. По этой причине, лучше посылать отчеты об ошибках в список рассылки.

В крайнем случае, посылайте отчеты об ошибках на бумаге по адресу:

```
GNU Debugger Bugs
Free Software Foundation Inc.
59 Temple Place - Suite 330
Boston, MA 02111-1307
USA
```

Основной принцип действенного составления отчетов об ошибках: **сообщайте все факты**. Если вы не уверены, оставить факт или исключить, оставьте его!

Часто люди опускают факты, потому что думают, что знают причины проблемы, и полагают, что некоторые детали не имеют значения. Таким образом, вы можете считать, что имя переменной, которую вы используете в примере, не имеет значения. Возможно это так, но нельзя быть уверенным в этом. Может быть ошибкой является неверное обращение к памяти, которое выбрало данные из ячеек, где хранилось это имя; возможно, если бы имя было другим, содержимое этих ячеек ввело бы отладчик в заблуждение, и ошибка не была бы замечена. Относитесь к этому осторожно, и приводите конкретные, полные примеры. Это самое простое что вы можете сделать, и наиболее полезное.

Помните, что цель отчета об ошибке состоит в том, чтобы дать нам возможность установить дефект. Может случиться, что об этой ошибке нам уже сообщали, но не вы, не мы не можем этого знать, если отчет об ошибке не будет полным и самодостаточным.

Иногда люди дают несколько поверхностных фактов и спрашивают, “не говорит ли это об ошибке?”. Такие сообщения о дефектах бесполезны, и мы убеждаем всех *отказываться отвечать на них*, за исключением того, чтобы побудить автора отчета послать его правильно.

Чтобы дать нам возможность устранить ошибку, вы должны включить в сообщение следующее:

- Версию GDB. GDB сообщает ее при вызове без параметров; вы можете также вывести ее в любой момент, используя `show version`.

  Без этого мы не будем знать, имеет ли смысл поиск ошибки в текущей версии отладчика.

- Тип машины, которой вы пользуетесь, название и номер версии операционной системы.

- Какой компилятор (и его версия) использовался при компиляции GDB. Например, “gcc–2.8.1”.

- Какой компилятор (и его версия) использовался для компиляции отлаживаемой программы—например “gcc–2.8.1”, или “HP92453-01 A.10.32.03 HP C Compiler”. Для gcc, вы можете использовать `gcc --version` чтобы получить эту информацию; для других компиляторов, смотрите их документацию.

- Параметры команды, которые вы дали компилятору для компиляции вашего примера, с которым вы наблюдали ошибку. Например, использовали ли вы ‘`-O`’? Для гарантии, что вы не пропустите что-нибудь важное, перечисляйте все. Копии ‘`Makefile`’ (или результата вызова `make`) достаточно.

  Если мы должны будем угадывать аргументы, мы возможно сделаем это неправильно и можем не столкнуться с ошибкой.

- Полный сценарий ввода, и все необходимые исходные файлы, которые воспроизведут ошибку.

- Описание наблюдаемого вами поведения, которое вы считаете ошибочным. Например, “Это приводит к фатальному сигналу”.

  Конечно, если ошибка состоит в получении GDB фатального сигнала, то мы, конечно, заметим это. Но если ошибкой является некорректный вывод, мы можем не заметить этого, если это не бросается в глаза. Вы также можете не дать нам возможности ошибиться.

Даже если ваша проблема заключается в фатальном сигнале, вы все же должны сообщить об этом явно. Предположим, происходит что-то странное, например, ваша копия GDB рассинхронизировалась, или вы столкнулись с ошибкой в библиотеке Си вашей системы. (Такое бывало!) Ваша копия может завершиться аварийно, а наша нет. Если вы предупредите нас об ожидаемой аварии, а в нашей системе этого не произойдет, мы будем знать, что ошибка произошла не из-за нас. Если вы нас не предупредите, мы не сможем сделать никаких выводов из наших наблюдений.

- Если вы хотите предложить внести изменения в исходные тексты GDB, присылайте нам контекстные изменения. Даже если вы желаете обсудить что-нибудь из исходных текстов, ссылайтесь по контексту, а не по номеру строки.

  Номера строк в наших исходных текстах разработки не будут соответствовать вашим. Ваши номера строк не дадут нам никакой полезной информации.

Вот некоторые вещи, не являющиеся обязательными:

- Описание контекста ошибки.

  Часто люди, сталкивающиеся с ошибкой, тратят много времени на исследования, какие изменения входного файла приведут к ее исчезновению, а какие на нее не влияют.

  Это часто занимает много времени и приносит мало пользы, потому что мы найдем ошибку посредством выполнения одного примера под управлением отладчика с точками останова, а не чистыми выводами из серии примеров. Мы рекомендуем вам сохранить это время для чего-нибудь другого.

  Конечно, если вы сможете найти более простой пример для отчета *вместо* первоначального, это будет удобнее для нас. Выделение ошибок в выводе будет проще, выполнение под управлением отладчика будет занимать меньше времени, и так далее.

  Однако, это упрощение не является жизненно важным; если вы не хотите делать этого, сообщайте об ошибке в любом случае и посылайте нам весь тестовый материал, который вы использовали.

- Заплата для ошибки.

  Заплата для исправления ошибки действительно поможет нам, если это хорошая заплата. Но не опускайте необходимую информацию, такую как тестовый пример, предполагая, что заплата это все, в чем мы нуждаемся. Мы можем обнаружить проблемы с вашей заплатой и решить устранить ошибку другим путем, или мы можем вообще не понять смысл вашей заплаты.

  Иногда для такой сложной программы, как GDB, очень трудно создать пример, который заставит программу следовать по определенному пути в процессе выполнения. Если вы не пришлете нам пример, мы не сможем сконструировать его сами, и таким образом не сможем проверить, что ошибка устранена.

  И если мы не сможем понять, какую ошибку вы пытаетесь исправить, или почему ваша заплата являются улучшением, мы не используем ее. Тестовый пример поможет нам во всем разобраться.

- Предположения, в чем состоит ошибка, или от чего она зависит.

  Такие предположения обычно неверны. Даже мы не можем сделать правильных предположений о такого рода вещах до запуска отладчика и выявления фактов.

21 Редактирование командной строки

Эта глава описывает основные возможности интерфейса редактирования командной строки GNU.

21.1 Введение в редактирование строк

Следующие параграфы описывают нотацию, используемую для обозначения нажатия клавиш.

Текст `C-k` читается как 'Control-K' и описывает знак, полученный нажатием клавиши ⟨k⟩ при нажатой клавише Control.

Текст `M-k` читается как 'Meta-K' и описывает знак, введенный нажатием клавиши ⟨k⟩ при нажатой клавише Meta (если у вас она имеется). На многих клавиатурах клавиша Meta отмечается с помощью ⟨ALT⟩. На клавиатурах с двумя клавишами, отмеченными ⟨ALT⟩ (обычно по разные стороны от пробела), ⟨ALT⟩ на левой стороне обычно устанавливается для работы клавишей Meta. Правый ⟨ALT⟩ также может быть сконфигурирован для работы Meta, или он может быть сконфигурирован как другой модификатор, например как клавиша Compose для ввода символов с акцентами.

Если у вас нет клавиши Meta или ⟨ALT⟩, или другой клавиши, работающей как Meta, идентичное нажатие клавиш можно получить нажав *сначала* ⟨ESC⟩, а затем ⟨k⟩. Эти процессы называются *метафикацией* клавиши ⟨k⟩.

Текст `M-C-k` читается как 'Meta-Control-k' и описывает знак, полученный посредством *метафикации* `C-k`.

Кроме того, некоторые клавиши имеют собственные имена. Именно, ⟨DEL⟩, ⟨ESC⟩, ⟨LFD⟩, ⟨SPC⟩, ⟨RET⟩ и ⟨TAB⟩ в этом тексте или в файле инициализации обозначают сами себя (см. Раздел 21.3 [Файл инициализации Readline], с. 232). Если на вашей клавиатуре нет клавиши ⟨LFD⟩, нажатие `C-j` приведет к вводу желаемого символа. Клавиша ⟨RET⟩ на некоторых клавиатурах может быть отмечена как ⟨Return⟩ или ⟨Enter⟩.

21.2 Взаимодействие с Readline

Часто во время интерактивного сеанса вы вводите длинную строку текста только для того, чтобы заметить, что первое слово набрано неправильно. Библиотека Readline дает вам набор команд для управления текстом во время ввода, позволяя вам лишь исправить опечатку, а не набирать заново большую часть строки. С помощью этих команд редактирования, вы перемещаете курсор в место, требующее исправления, и удаляете или вставляете текст для коррекции. Затем, когда строка полностью исправлена, вы просто нажимаете ⟨RET⟩. Чтобы нажать ⟨RET⟩ вам не обязательно находиться в конце строки; вся строка вводится независимо от расположения в ней курсора.

21.2.1 Сведения первой необходимости

Для того, чтобы ввести знак в строку, просто нажмите его. Введенный знак появляется там, где был курсор, и затем курсор перемещается на одну позицию вправо. Если вы неверно набрали знак, вы можете использовать ваш знак уничтожения, чтобы вернуться и удалить неверный знак.

Иногда вы можете набрать символ ошибочно, и не заметить ошибки, пока не напечатаете несколько других знаков. В этом случае, вы можете набрать `C-b`, чтобы переместить курсор влево, и затем исправить вашу ошибку. После этого, вы можете переместить курсор вправо нажатием `C-f`.

Когда вы добавляете текст в середину строки, символы справа от курсора ‘сдвигаются вперед’, чтобы освободить место для вставляемого текста. Аналогично, когда вы удаляете текст за курсором, символы справа от него ‘сдвигаются назад’, занимая пустое пространство, созданное в результате удаления текста. Ниже следует список команд первой необходимости для редактирования вводимого текста.

`C-b` Переместиться назад на одну позицию.

`C-f` Переместиться вперед на одну позицию.

⟨DEL⟩ или ⟨Backspace⟩
Удалить символ слева от курсора.

`C-d` Удалить символ под курсором.

Ввод символов
Вставить символ в строку в позицию курсора.

`C-_` или `C-x C-u`
Отменить последнюю команду редактирования. Вы можете отменить все, вернувшись назад к пустой строке.

(В зависимости от вашей конфигурации, клавиша ⟨Backspace⟩ может быть настроена удалять знак слева от курсора, а ⟨DEL⟩ удалять знак под курсором, как `C-d`, вместо символа, расположенного от курсора слева.)

21.2.2 Команды перемещения Readline

Приведенная выше таблица описывает только самые базовые последовательности клавиш, которые могут вам понадобиться для редактирования строки ввода. Для удобства, в дополнение к `C-b`, `C-f`, `C-d` и ⟨DEL⟩ были добавлены многие другие команды. Вот некоторые из них, предназначенные для более быстрого перемещения по строке.

`C-a` Переместиться в начало строки.

`C-e` Переместиться в конец строки.

`M-f` Переместиться вперед на слово. Слово состоит из букв и цифр.

`M-b` Переместиться назад на слово.

`C-l` Очистить экран, напечатав текущую строку заново вверху экрана.

Заметьте, что `C-f` перемещает курсор вперед на символ, в то время как `M-f` перемещает вперед на слово. Это своего рода соглашение, что при нажатии клавиши Control производятся действия над символами, при нажатии клавиши Meta—над словами.

21.2.3 Команды уничтожения Readline

Уничтожение текста означает уничтожение текста из строки, но сохранение его для дальнейшего использования, обычно для *восстановления* (повторной вставки) обратно в строку. (‘Вырезать’ и ‘вставить’ являются более свежими жаргонными синонимами для ‘уничтожить’ и ‘восстановить’.)

Если в описании команды сказано, что она ‘уничтожает’ текст, то вы можете быть уверены, что позже его можно будет получить обратно в другом (или том же самом) месте.

Когда вы используете команду уничтожения, текст сохраняется в *кольцевом списке уничтожений*. Любое число последовательных уничтожений сохраняет весь уничтоженный текст вместе, так что когда вы восстанавливаете его назад, вы получите все. Список уничтожений не имеет привязки к строкам; текст, уничтоженный вами в предыдущей строке ввода, доступен для восстановления позже, когда вы вводите другую строку.

Вот список команд для уничтожения текста.

`C-k` Уничтожить текст от текущей позиции курсора до конца строки.

`M-d` Уничтожить от курсора до конца текущего слова, или, если курсор находится между словами, до конца следующего слова. Границы слов такие же, как и используемые `M-f`.

`M-`⟨DEL⟩ Уничтожить от курсора до начала текущего слова, или, если курсор находится между словами, до начала предыдущего слова. Границы слов такие же, как и используемые `M-b`.

`C-w` Уничтожить от курсора до предыдущего пробельного символа. Это отличается от M-⟨DEL⟩, так как границы слова различаются.

Вот как можно *восстановить* текст обратно в строку. Восстановление означает копирование последнего уничтоженного текста из буфера уничтожений.

`C-y` Восстановить последний уничтоженный текст в буфер перед курсором.

`M-y` Циклический сдвиг по кольцевому списку уничтожений, и восстановление новой вершины. Вы можете использовать это, только если предыдущая команда была `C-y` или `M-y`.

21.2.4 Параметры команд Readline

Вы можете передавать числовые параметры командам Readline. Иногда параметр действует как счетчик повторений, иногда он является *знаком*, если аргумент имеет знак. Если вы передаете отрицательный параметр команде, которая обычно действует в прямом направлении, то она будет действовать в обратном направлении. Например, чтобы уничтожить текст до начала строки, вы можете набрать '`M- C-k`'.

Общий способ передачи числовых параметров команде состоит в наборе Meta-цифр перед командой. Если первая набранная 'цифра' есть знак минус ('`-`'), тогда знак аргумента будет отрицательным. Если вы набрали одну мета-цифру для начала параметра, вы можете набрать оставшиеся цифры, и потом команду. Например, чтобы передать команде `C-d` параметр 10, вы можете набрать '`M-1 0 C-d`'.

21.2.5 Поиск команд в истории

Readline предоставляет команды для поиска в истории команд строк, содержащих указанную подстроку. Существует два режима поиска: *наращиваемый* и *ненаращиваемый*.

Наращиваемый поиск начинается до того, как пользователь закончит ввод строки поиска. По мере ввода очередных символов строки поиска, Readline отображает следующий элемент из истории, соответствующий строке, введенной на данный момент. Наращиваемый поиск требует ровно столько символов, сколько требуется для нахождения желаемого элемента истории. Для поиска определенной строки в истории в обратном направлении, введите `C-r`. Ввод `C-s` производит поиск в прямом направлении. Символы, присутствующие в значении переменной `isearch-terminators`, используются для завершения наращиваемого поиска. Если этой переменной не было присвоено значение, знаки ⟨ESC⟩ и `C-J` будут завершать наращиваемый поиск. `C-g` прерывает наращиваемый поиск и восстанавливает исходную строку. Когда поиск завершается, элемент истории, содержащий искомую строку, становится текущей строкой.

Для нахождения других подходящих элементов списка истории, введите соответственно `C-r` или `C-s`. Это произведет поиск вперед или назад в истории до следующего элемента, соответствующего введенной строке поиска. Любая другая последовательность клавиш, привязанная к команде Readline, завершит поиск и выполнит эту команду. Например, ⟨RET⟩ завершит поиск и примет эту строку, таким образом выполняя команду из списка истории.

Ненаращиваемый поиск считывает строку поиска целиком, до начала поиска соответствующих строк истории. Строка поиска может быть введена пользователем или являться частью содержимого текущей строки.

21.3 Файл инициализации Readline

Хотя библиотека Readline поставляется с установленным по умолчанию набором привязок клавиш, аналогичному Emacs, возможно использование другого набора привязок. Любой пользователь может настраивать программы, которые используют Readline, помещая команды в файл *inputrc*, обычно в своем домашнем каталоге. Имя этого файла берется из переменной среды `INPUTRC`. Если эта переменная не установлена, по умолчанию берется файл '`~/.inputrc`'.

Когда запускается программа, использующая библиотеку Readline, файл инициализации считывается и устанавливаются привязки клавиш.

Кроме того, команда `C-x C-r` считывает файл инициализации заново, так что изменения, которые вы могли сделать к этому времени, вступают в силу.

21.3.1 Синтаксис файла инициализации Readline

Существуют всего несколько конструкций, которые допускаются в файле инициализации Readline. Пустые строки игнорируются. Строки, начинающиеся с '`#`', являются комментариями. Строки, начинающиеся с '`$`', обозначают условные конструкции (см. Раздел 21.3.2 [Условные конструкции инициализации], с. 235). Другие строки обозначают установку переменных и привязки клавиш.

Установка переменных

Вы можете изменять поведение Readline во время выполнения посредством изменения значений переменных, используя команду `set` в файле инициализации. Вот как можно изменить привязку клавиш Emacs, используемую по умолчанию, для использования команд редактирования строки `vi`:

```
set editing-mode vi
```

Основная часть поведения при выполнении изменяется с помощью следующих переменных.

`bell-style`
: Контролирует, что происходит, когда Readline хочет издать звук на терминале. Если установлено в '`none`', Readline никогда не издает звук. Если установлено в '`visible`', Readline использует визуальный звонок, если есть возможность. Если установлено в '`audible`' (по умолчанию), Readline пытается издать звук на терминале.

`comment-begin`
: Строка для вставки в начало строки, когда выполняется команда '`insert-comment`'. По умолчанию `"#"`.

`completion-ignore-case`
: Если установлено в '`on`', Readline производит проверку совпадений и завершение имени файла без учета регистра. По умолчанию '`off`'.

`completion-query-items`
: Количество возможных завершений, определяющее, когда у пользователя запрашивается, хочет ли он увидеть список возможных вариантов. Если число возможных завершений больше этого значения, Readline спросит у пользователя, хочет он их просмотреть или нет; в противном случае, они просто отображаются. По умолчанию устанавливается предел `100`.

`convert-meta`
: Если установлено в ‘`on`’, Readline будет преобразовывать символы с установленным восьмым битом в последовательность клавиш ASCII, удаляя восьмой бит и подставляя в качестве префикса знак ⟨ESC⟩, тем самым преобразовывая их в последовательность клавиш с Meta-префиксом. По умолчанию ‘`on`’.

`disable-completion`
: Если установлено в ‘`on`’, Readline будет препятствовать завершению слов. Знаки завершения будут вставляться в строку так, как если бы они отображались в `self-insert`. По умолчанию ‘`off`’.

`editing-mode`
: Переменная `editing-mode` контролирует, какой набор привязок клавиш используется. По умолчанию, Readline запускается в режиме редактирования Emacs, где нажатия клавиш очень похожи на Emacs. Эта переменная может быть установлена или в ‘`emacs`’, или в ‘`vi`’.

`enable-keypad`
: Когда установлено в ‘`on`’, Readline будет пытаться активизировать малую клавиатуру приложения, когда она вызывается. Это требуется некоторым системам для активации клавиш со стрелками. По умолчанию ‘`off`’.

`expand-tilde`
: Если установлено в ‘`on`’, производится раскрывание тильды, когда Readline осуществляет завершение слова. По умолчанию ‘`off`’.

`horizontal-scroll-mode`
: Эта переменная может быть установлена в ‘`on`’ или ‘`off`’. Установка в ‘`on`’ означает, что текст в редактируемых строках будет прокручиваться горизонтально в одной строке экрана, когда ширина строки становится больше ширины экрана, вместо переноса на новую строку. По умолчанию, эта переменная установлена в ‘`off`’.

`input-meta`
: Если установлено в ‘`on`’, Readline включит восьмибитовый ввод (восьмой бит не будет удаляться из считываемых символов) независимо от того, поддерживает ли это терминал. Значение по умолчанию ‘`off`’. Имя `meta-flag` является синонимом для этой переменной.

`isearch-terminators`
: Строка из знаков, которые должны прекращать наращиваемый поиск без последующего выполнения знака как команды (см. Раздел 21.2.5 [Поиск в истории], с. 231). Если этой переменной не присвоено значение, наращиваемый поиск прекращают знаки ⟨ESC⟩ и *C-J*.

`keymap`
: Устанавливает текущую раскладку клавиатуры Readline для привязок команд к клавишам. Возможные имена для `keymap`: `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-command` и `vi-insert`. `vi` эквивалентно `vi-command`; `emacs` эквивалентно `emacs-standard`. Значение по умолчанию `emacs`. Значение переменной `editing-mode` также влияет на раскладку по умолчанию.

`mark-directories`
: Если установлено в ‘`on`’, к именам каталогов после завершения добавляется косая черта. По умолчанию ‘`on`’.

`mark-modified-lines`
: Эта переменная, будучи установлена в ‘`on`’, велит Readline отображать звездочку (‘`*`’) в начале тех строк истории, которые были модифицированы. По умолчанию, эта переменная установлена в ‘`off`’.

`output-meta`
: Если установлено в ‘`on`’, Readline будет отображать знаки с установленным восьмым битом непосредственно, а не в виде экранирующих последовательностей с Meta-префиксом. По умолчанию ‘`off`’.

`print-completions-horizontally`
: Если установлено в ‘`on`’, Readline будет отображать завершения, отсортированные горизонтально в алфавитном порядке, а не вниз по экрану. По умолчанию ‘`off`’.

`show-all-if-ambiguous`
: Это изменяет поведение по умолчанию функций завершения. При установке в ‘`on`’, если слово имеет более одного возможного завершения, они будут выводиться немедленно, вместо подачи сигнала. По умолчанию ‘`off`’.

`visible-stats`
: Если установлено в ‘`on`’, при выводе возможных завершений, к имени файла добавляется знак, обозначающий тип файла. По умолчанию ‘`off`’.

Привязки клавиш
: Синтаксис для управления привязками клавиш в файле инициализации прост. Во-первых, вы должны найти имя команды, которую вы хотите изменить. Следующий раздел содержит таблицы с именем команды, привязкой клавиш по умолчанию, если таковые есть, и коротким описанием, что делает команда.

  Если вы знаете имя команды, просто поместите в строке файла инициализации название клавиши, к которой вы хотите привязать команду, двоеточие, и затем имя команды. Название клавиши может быть выражено различными способами, в зависимости от того, как вам удобнее.

  назв-клавиши: *имя-функции* или *макрос*
  : *назв-клавиши*—это название клавиши, записанное по-английски. Например:

    ```
    Control-u: universal-argument
    Meta-Rubout: backward-kill-word
    Control-o: "> output"
    ```

    В этом примере, *`C-u`* привязана к функции `universal-argument`, а *`C-o`* привязана к выполнению макрокоманды, записанной с правой стороны (то есть, вставить текст ‘`> output`’ в строку).

  "*послед-клавиш*": *имя-функции* или *макро*
  : *послед-клавиш* отличается от вышеупомянутого *назв-клавиши* тем, что позволяет определять строки, обозначающие целую последовательность клавиш, посредством ее заключения в двойные кавычки. Могут быть использованы некоторые экранирующие последовательности в стиле GNU Emacs, как в следующем примере, но имена специальных знаков не распознаются.

    ```
    "\C-u": universal-argument
    "\C-x\C-r": re-read-init-file
    ```

```
"\e[11~": "Function Key 1"
```

В этом примере, *C-u* привязывается к функции `universal-argument` (как это было в первом примере), '*C-x C-r*' привязывается к функции `re-read-init-file` и '⟨ESC⟩ *[1 1 ~*' привязывается к вставке текста '`Function Key 1`'.

Следующие экранирующие последовательности в стиле GNU Emacs доступны при определении последовательности клавиш:

`\C-`	префикс Control
`\M-`	префикс Meta
`\e`	префикс экранирующего знака
`\\`	обратная косая черта
`\"`	⟨"⟩, знак двойных кавычек
`\'`	⟨'⟩, одинарная кавычка или апостроф

В дополнение к экранирующим последовательностям стиля GNU Emacs, доступен второй набор последовательностей с обратной косой чертой:

`\a`	тревога (звуковой сигнал)
`\b`	переместиться назад на одну позицию
`\d`	удаление
`\f`	перевод страницы
`\n`	новая строка
`\r`	возврат каретки
`\t`	горизонтальная табуляция
`\v`	вертикальная табуляция
`\`*nnn*	знак, восьмеричное значение кода `ASCII` которого есть *nnn* (от одной до трех цифр)
`\x`*nnn*	знак, шестнадцатеричное значение кода `ASCII` которого есть *nnn* (от одной до трех цифр)

При вводе текста макрокоманды, для обозначения ее определения должны использоваться одиночные или двойные кавычки. Предполагается, что текст без кавычек является именем функции. В теле макрокоманды, экранирующие последовательности с обратной косой чертой раскрываются. Обратная косая черта будет экранировать любой другой знак в тексте макрокоманды, включая '"' и ''''. Например, следующая привязка велит '*C-x* \' вставлять одиночную '\' в строку:

```
"\C-x\\": "\\"
```

21.3.2 Условные конструкции инициализации

Readline реализует возможности, аналогичные по смыслу возможностям условной компиляции препроцессора Си, позволяющие производить привязки клавиш и установку переменных в результате тестов. Вот четыре директивы, используемые анализатором.

`$if` Конструкция `$if` позволяет производить привязки в зависимости от режима редактирования, используемого терминала, или приложения, использующего Readline. Содержимое теста продолжается до конца строки; для его ограничения не требуются никакие знаки.

`mode` Форма `mode=` директивы `$if` используется для проверки в каком из режимов находится Readline: `emacs` или `vi`. Это может быть использовано, например, вместе с командой '`set keymap`' для установки привязок в наборы `emacs-standard` и `emacs-ctlx`, только если Readline запускается в режиме `emacs`.

`term` Форма `term=` может использоваться для включения привязок клавиш, уникальных для какого-либо терминала, возможно для привязки вывода последовательности клавиш к функциональным клавишам терминала. Слово, стоящее справа от '`=`', сравнивается как с полным названием терминала, так и с частью названия, идущей до первого '`-`'. Это позволяет, например, опознавать как `sun`, так и `sun-cmd`.

`application`
Конструкция *application* используется для включения установок, уникальных для какого-либо приложения. Каждая программа, использующая библиотеку Readline, устанавливает *application name*, и вы можете проверить его. Это может быть использовано для привязки последовательностей клавиш к функциям, полезным в конкретной программе. Например, следующая команда добавляет последовательность, которая заключает в кавычки текущее или предыдущее слово в Bash:

```
$if Bash
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif
```

`$endif` Эта команда, как показано в предыдущем примере, заканчивает команду `$if`.

`$else` Команды этой ветви директивы `$if` выполняются, если проверка заканчивается неудачей.

`$include` Эта директива принимает в качестве аргумента одно имя файла и считывает из него команды и привязки клавиш.

```
$include /etc/inputrc
```

21.3.3 Пример файла инициализации

Вот пример файла *inputrc*. Он иллюстрирует привязки клавиш, присвоение значений переменным и синтаксис условий.

```
# Этот файл управляет поведением редактирования строки ввода в
# программах, использующих библиотеку Gnu Readline.  Среди таких программ
# FTP, Bash, и Gdb.
#
# Вы можете заново считать файл inputrc с помощью C-x C-r.
# Строки, начинающиеся с '#', являются комментариями.
#
# Сначала, включим все общесистемные привязки и переменные из
# /etc/Inputrc
$include /etc/Inputrc

#
# Установка различных привязок для режима emacs.

set editing-mode emacs

$if mode=emacs

Meta-Control-h:  backward-kill-word Текст после имени функции игнорируется

#
# Стрелки в режиме малой клавиатуры
#
#"\M-OD":        backward-char
#"\M-OC":        forward-char
#"\M-OA":        previous-history
#"\M-OB":        next-history
#
# Стрекли в режиме ANSI
#
"\M-[D":        backward-char
"\M-[C":        forward-char
"\M-[A":        previous-history
"\M-[B":        next-history
#
# Стрелки в восьмибитном режиме малой клавиатуры
#
#"\M-\C-OD":       backward-char
#"\M-\C-OC":       forward-char
#"\M-\C-OA":       previous-history
#"\M-\C-OB":       next-history
#
# Стрелки в восьмибитовом режиме ANSI
#
#"\M-\C-[D":       backward-char
#"\M-\C-[C":       forward-char
#"\M-\C-[A":       previous-history
#"\M-\C-[B":       next-history

C-q: quoted-insert
```

```
$endif

# Привязки старого стиля.  Устанавливается по умолчанию.
TAB: complete

# Макрокоманды, удобные при взаимодействии с оболочкой
$if Bash
# редактирование пути
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# Подготовка к вводу слова в кавычках - вставляет открывающуюся и
# закрывающуюся двойные кавычки и помещает курсор сразу за открывающей
"\C-x\"": "\"\"\C-b"
# вставляет обратную косую черту (testing backslash escapes in sequences
# and macros)  "\C-x\\": "\\"
# Заключает в кавычки текущее или предыдущее слово
"\C-xq": "\eb\"\ef\""
# Добавляет привязку для обновления строки, которая непривязана
"\C-xr": redraw-current-line
# Редактирование переменной в текущей строке
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# использовать визуальный звонок, если он доступен
set bell-style visible

# не урезать символы при чтении до 7 бит
set input-meta on

# позволяет ввод символов iso-latin1, вместо из преобразования к
# последовательностям с Meta-префиксом
# prefix-meta sequences
set convert-meta off

# отображает символы непосредственно с установленным восьмым битом, а не
# в виде знаков с Meta-префиксом
set output-meta on

# если существует более 150 возможных завершений слова, запросить
# пользователя, хочет ли он видеть их все
set completion-query-items 150

# Для FTP
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg
$endif
```

21.4 Привязываемые команды Readline

Этот раздел описывает команды Readline, которые могут быть привязаны к последовательностям клавиш.

21.4.1 Команды для перемещения

`beginning-of-line (C-a)`
: Переместиться в начало текущей строки.

`end-of-line (C-e)`
: Переместиться в конец строки.

`forward-char (C-f)`
: Переместиться на один знак вперед.

`backward-char (C-b)`
: Переместиться назад на один знак.

`forward-word (M-f)`
: Переместиться вперед до конца следующего слова. Слова составляются из букв и цифр.

`backward-word (M-b)`
: Переместиться назад к началу текущего или предыдущего слова. Слова составляются из букв и цифр.

`clear-screen (C-l)`
: Очистить экран и перерисовать текущую строку, оставляя ее наверху экрана.

`redraw-current-line ()`
: Обновить текущую строку. По умолчанию не привязана.

21.4.2 Команды для манипуляции историей

`accept-line (Newline, Return)`
: Ввод строки независимо от положения курсора. Если строка непуста, добавить ее к списку истории. Если эта строка была строкой истории, то восстановить строку истории до ее первоначального состояния.

`previous-history (C-p)`
: Переместиться ‘вверх’ по списку истории.

`next-history (C-n)`
: Переместиться ‘вниз’ по списку истории.

`beginning-of-history (M-<)`
: Переместиться к первой строке истории.

`end-of-history (M->)`
: Переместиться в конец истории ввода, то есть к строке, которая сейчас редактируется.

`reverse-search-history (C-r)`
: Обратный поиск начиная с текущей строки и перемещаясь по мере необходимости ‘вверх’ по истории. Это наращиваемый поиск.

`forward-search-history (C-s)`
: Прямой поиск начиная с текущей строки и перемещаясь по мере необходимости ‘вниз’ по истории. Это наращиваемый поиск.

`non-incremental-reverse-search-history (M-p)`
: Обратный поиск начиная с текущей строки и перемещаясь по мере необходимости ‘вверх’ по истории, используя ненаращиваемый поиск строки, заданной пользователем.

`non-incremental-forward-search-history (M-n)`
: Прямой поиск начиная с текущей строки и перемещаясь по мере необходимости ‘вниз’ по истории, используя ненаращиваемый поиск строки, заданной пользователем.

`history-search-forward ()`
: Прямой поиск в истории строки символов между началом текущей строки и точкой. Это ненаращиваемый поиск. По умолчанию эта команда не привязана.

`history-search-backward ()`
: Обратный поиск в истории строки символов между началом текущей строки и точкой. Это ненаращиваемый поиск. По умолчанию эта команда не привязана.

`yank-nth-arg (M-C-y)`
: Вставить первый аргумент предыдущей команды (обычно второе слово предыдущей строки). С аргументом *n*, вставляет *n*-ное слово из предыдущей команды (слова в предыдущей команде начинаются со слова 0). Отрицательный аргумент вставляет *n*-ное с конца слово предыдущей команды.

`yank-last-arg (M-., M-_)`
: Вставить последний аргумент предыдущей команды (последнее слово предыдущего элемента истории). С аргументом, ведет себя точно так же, как `yank-nth-arg`. Последовательные вызовы `yank-last-arg` перемещают назад по списку истории, вставляя последний аргумент каждой строки по очереди.

21.4.3 Команды для изменения текста

`delete-char (C-d)`
: Удалить символ в позиции курсора. Если курсор находится в начале строки, в строке нет символов и последний набранный знак не был привязан к `delete-char`, возвращает EOF.

`backward-delete-char (Rubout)`
: Удалить символ за курсором. Числовой параметр предписывает уничтожать символы, а не удалять их.

`forward-backward-delete-char ()`
: Удалить символ под курсором, если не находится в конце строки, в этом случае удаляется символ за курсором. По умолчанию, эта функция не привязана к какой-либо клавише.

`quoted-insert (C-q, C-v)`
: Добавить в строку следующий введенный знак, каким бы он ни был. Таким образом можно вставить последовательность клавиш, например *C-q*.

`tab-insert (M-TAB)`
: Вставить знак табуляции.

`self-insert (a, b, A, 1, !, ...)`
: Эти символы вставляют сами себя.

`transpose-chars (C-t)`
: Переместить знак перед курсором вперед за знак под курсором, также перемещая курсор вперед. Если точка вставки находится в конце строки, то переставить последние два знака в строке. Отрицательные аргументы не оказывают действия.

`transpose-words (M-t)`
: Переместить слово перед точкой за слово после точки. Также перемещает точку за это слово.

`upcase-word (M-u)`
: Перевести в верхний регистр текущее (или следующее) слово. С отрицательным аргументом, переводит в верхний регистр предыдущее слово, но не перемещает курсор.

`downcase-word (M-l)`
: Перевести в нижний регистр текущее (или следующее) слово. С отрицательным аргументом переводит в нижний регистр предыдущее слово, но не перемещает курсор.

`capitalize-word (M-c)`
: Перевести текущее (или предыдущее) слово в нижний регистр с первой заглавной буквой. С отрицательным аргументом, действует на предыдущее слово, но не перемещает курсор.

21.4.4 Уничтожение и восстановление

`kill-line (C-k)`
: Уничтожить текст от точки до конца строки.

`backward-kill-line (C-x Rubout)`
: Уничтожить назад до начала строки.

`unix-line-discard (C-u)`
: Уничтожить назад от курсора до начала текущей строки.

`kill-whole-line ()`
: Уничтожить все символы в текущей строке, независимо от позиции точки. По умолчанию эта функция не привязана.

`kill-word (M-d)`
: Уничтожить от точки до конца текущего слова, или, если курсор находится между словами, до конца следующего слова. Границы слова такие же, как для `forward-word`.

`backward-kill-word (M-DEL)`
: Уничтожить слово за точкой. Границы слова такие же, как для `backward-word`.

`unix-word-rubout (C-w)`
: Уничтожает слово за точкой, используя в качестве границы слова пробелы. Уничтоженный текст сохраняется в кольцевом списке уничтожений.

`delete-horizontal-space ()`
: Удалить все пробелы и знаки табуляции вокруг точки. По умолчанию эта функция не привязана.

`kill-region ()`
: Уничтожить текст в текущей области. Эта функция по умолчанию не привязана.

`copy-region-as-kill ()`
: Копировать текст области в буфер уничтожения, так что он может быть сразу восстановлен. Эта команда по умолчанию не привязана.

`copy-backward-word ()`
: Копировать слово перед точкой в буфер уничтожений. Границы слова такие же, как для `backward-word`. Эта команда по умолчанию не привязана.

`copy-forward-word ()`
: Копировать слово, следующее за точкой, в буфер уничтожений. Границы слова такие же, как для `forward-word`. Эта команда по умолчанию не привязана.

`yank (C-y)`
: Восстановить вершину кольцевого списка уничтожений в буфер в текущую позицию курсора.

`yank-pop (M-y)`
: Циклический сдвиг кольцевого списка уничтожений и восстановление новой вершины. Вы можете использовать эту команду только если предыдущей командой была `yank` или `yank-pop`.

21.4.5 Определение числовых параметров

`digit-argument (`*`M-0`*`, `*`M-1`*`, ... `*`M-`*`)`
: Добавить эту цифру к вводимому аргументу, или начать новый аргумент. *M-* начинает отрицательный аргумент.

`universal-argument ()`
: Это другой способ задать аргумент. Если за этой командой следуют одна или несколько цифр, возможно со знаком минус в начале, то они определяют аргумент. Если за командой следуют цифры, повторное выполнение `universal-argument` заканчивает числовой аргумент, а в противном случае он игнорируется. В особом случае, если непосредственно за этой командой следует или цифра, или знак минус, аргумент для следующей команды умножается на четыре. Изначально аргумент равен единице, так что выполнение этой функции первый раз делает его равным четырем, второй раз—шестнадцати, и так далее. По умолчанию эта функция не привязана к клавише.

21.4.6 Readline вводит за вас

`complete (⟨TAB⟩)`
: Пытается завершить текст перед курсором. Действие функции зависит от приложения. Обычно, если вы вводите имя файла, вы можете выполнить завершение имени файла; если вы вводите команду, вы можете завершить команду; если вводите символ для GDB, можете выполнить завершение имени символа; если вы вводите переменную для Bash, можете завершить имя переменной, и так далее.

`possible-completions (M-?)`
: Перечислить возможные завершения текста перед курсором.

`insert-completions (M-*)`
: Вставить все завершения текста перед точкой, которые были бы созданы `possible-completions`.

`menu-complete ()`
: Аналогично `complete`, но замещает завершаемое слово одним совпадением из списка возможных завершений. Повторяемое выполнение `menu-complete` обходит список возможных завершений, вставляя каждое совпадение по очереди. В конце списка завершений издается звуковой сигнал и восстанавливается исходный текст. Аргумент *n* перемещает на *n* позиций вперед в списке совпадений; отрицательный аргумент может использоваться для перемещения по списку назад. Эта команда предназначается для привязки к ⟨TAB⟩, но по умолчанию не привязана.

`delete-char-or-list ()`
: Удаляет знак в позиции курсора, если он не находится в начале или конце строки (как `delete-char`). Если курсор находится в конце строки, поведение аналогично `possible-completions`. Эта команда по умолчанию не привязана.

21.4.7 Клавиатурные макросы

`start-kbd-macro (C-x ()`
: Начать сохранение вводимых символов в текущий клавиатурный макрос.

`end-kbd-macro (C-x ))`
: Прекратить сохранение вводимых символов в текущий клавиатурный макрос и запомнить его определение.

`call-last-kbd-macro (C-x e)`
: Выполнить последний определенный клавиатурный макрос, выводя символы макроса так, как если бы он набирался на клавиатуре.

21.4.8 Некоторые другие команды

`re-read-init-file (C-x C-r)`
: Считать содержание файла *inputrc*, и подключить любые найденные там привязки клавиш и присвоения переменных.

`abort (C-g)`
: Прервать текущую команду редактирования и издать звук на терминале (зависит от установки `bell-style`).

`do-uppercase-version (M-a, M-b, M-`*x*`, ...)`
: Если метафицированный знак *x* находится в нижнем регистре, выполнить команду, привязанную к соответствующему символу в верхнем регистре.

`prefix-meta (`⟨ESC⟩`)`
: Метафицировать следующий введенный вами символ. Это для клавиатур, не имеющих клавиши Meta. Ввод ‘⟨ESC⟩ `f`’ эквивалентен вводу *`M-f`*.

`undo (C-_, C-x C-u)`
: Наращиваемая отмена, запоминаемая отдельно для каждой строки.

`revert-line (M-r)`
: Отменить все изменения, сделанные в этой строке. Это аналогично выполнению достаточного числа раз команды `undo`, чтобы вернуться в начало.

`tilde-expand (M-~)`
: Выполнить раскрывание знака тильды для текущего слова.

`set-mark (C-@)`
: Установить отметку в текущей точке. Если задан числовой аргумент, отметка устанавливается в указанной позиции.

`exchange-point-and-mark (C-x C-x)`
: Поменять местами точку и отметку. Текущая позиция курсора устанавливается в сохраненную позицию, а старое положение курсора запоминается как отметка.

`character-search (C-])`
: Считывается символ, и точка перемещается к следующему появлению этого символа. Отрицательный аргумент производит поиск вхождения символа в обратном направлении.

`character-search-backward (M-C-])`
: Считывается символ, и точка перемещается к предыдущему появлению этого символа. Отрицательный аргумент производит поиск последующих вхождений символа.

`insert-comment (M-#)`
: Значение переменной `comment-begin` вставляется в начало текущей строки, и строка вводится, так как если бы был введен знак ввода строки.

`dump-functions ()`
: Напечатать все функции и их привязки к клавишам в выходной поток Readline. Если задан числовой аргумент, вывод форматируется так, что он может быть сделан частью файла *inputrc*. По умолчанию, эта команда не привязана.

`dump-variables ()`
: Напечатать все устанавливаемые переменные и их значения в выходной поток Readline. Если задан числовой аргумент, вывод форматируется так, что он может быть сделать частью файла *inputrc*. Эта команда по умолчанию не привязана.

`dump-macros ()`
: Напечатать все последовательности клавиш Readline, привязанные к макросам, и выводимые ими строки. Если задан числовой аргумент, вывод форматируется так, что он может быть сделан частью файла *inputrc*. Эта команда по умолчанию не привязана.

21.5 Режим vi Readline

Хотя библиотека Readline не имеет полного набора функций редактирования `vi`, она все же содержит достаточно для простого редактирования строки. Режим `vi` Readline функционирует так, как определено в стандарте POSIX 1003.2.

Для интерактивного переключения между режимами редактирования `emacs` и `vi`, используйте команду *M-C-j* (`toggle-editing-mode`). По умолчанию, Readline использует режим `emacs`.

Когда вы вводите строку в режиме `vi`, вы уже находитесь в режиме ‘вставки’, как если бы вы набрали ‘`i`’. Нажатие ⟨ESC⟩ переключает вас в ‘командный’ режим, в котором вы можете редактировать текст строки стандартными клавишами перемещения `vi`, перемещаться к предыдущим строкам истории посредством ‘`k`’ и к последующим строкам посредством ‘`j`’, и так далее.

22 Использование истории в интерактивном режиме

Эта глава описывает, как использовать библиотеку GNU History интерактивно, с точки зрения пользователя. Она должна рассматриваться как руководство пользователя.

22.1 Раскрывание истории

Библиотека History обеспечивает средства раскрывания истории, подобные предоставляемым `csh`. Этот раздел описывает синтаксис, использующийся для управления информацией истории.

Раскрывание истории вводит слова из списка истории во входной поток, облегчая повторение команд, вставку аргументов предыдущей команды в текущую строку ввода, или быстрое устранение ошибок в предыдущей команде.

Раскрывание истории происходит в два этапа. Первый заключается в определении, какая из строк списка истории должна использоваться в процессе замены. На втором этапе выбираются части этой строки для включения в текущую. Строка, выбранная из истории, называется *событием*, а использованные части называются *словами*. Существуют различные *модификаторы* для манипулирования выбранными словами. Строка разбивается на слова так же, как это делает Bash, так что несколько слов, заключенных в кавычки, рассматриваются как одно слово. Раскрывания истории вводятся появлением знака раскрывания истории, по умолчанию '`!`'.

22.1.1 Указатели событий

Указатель событий является ссылкой на запись командной строки в списке истории.

`!` Начать подстановку истории, если только за `!` не следует пробел, знак табуляции, знак конца строки, '`=`' или '`(`'.

`!`*n* Сослаться на командную строку с номером *n*.

`!-`*n* Сослаться на командную строку, отстоящую на *n* строк назад.

`!!` Сослаться на предыдущую команду. Это синоним для '`!-1`'.

`!`*строка* Сослаться на самую последнюю команду, начинающуюся со *строки*.

`!?`*строка*`[?]`
Сослаться на самую последнюю команду, содержащую *строку*. '`?`' в конце может быть опущен, если знак новой строки следует немедленно за *строкой*.

`^`*строка1*`^`*строка2*`^`
Быстрая подстановка. Повторяет последнюю команду, заменяя *строку1* на *строку2*. Эквивалентно `!!:s/`*строка1*`/`*строка2*`/`.

`!#` Вся командная строка, введенная до этого момента.

22.1.2 Указатели слов

Указатели слов используются для выбора желаемых слов из события. Спецификация события отделяется от указателя слова двоеточием. Его можно опустить, если указатель слова начинается с '`^`', '`$`', '`*`', '`-`' или '`%`'. Слова нумеруются от начала строки, причем первому слову присваивается номер 0 (ноль). Слова вставляются в текущую строку, разделенные одиночными пробелами.

Например,

`!!` обозначает предыдущую команду. Когда вы это вводите, предыдущая команда повторяется один к одному.

`!!:$` обозначает последний аргумент предыдущей команды. Это может быть сокращено как `!$`.

`!fi:2` обозначает второй аргумент самой последней команды, начинавшейся с букв `fi`.

Вот указатели слов:

`0 (ноль)` Нулевое слово. Для многих приложений, это командное слово.

n *n*-ное слово.

`^` Первый аргумент, то есть слово 1.

`$` Последний аргумент.

`%` Слово, соответствующее самому последнему поиску '`?`*строка*`?`'.

x`-`*y* Диапазон слов; '`0-`*y*' сокращается как '`-`*y*'.

`*` Все слова, за исключением нулевого. Это синоним для '`1-$`'. Даже если в событии имеется только одно слово, использование '`*`' не является ошибкой; в этом случае возвращается пустая строка.

x`*` Сокращение для '*x*`-$`'

x`-` Сокращает '*x*`-$`' как '*x*`*`', но опускает последнее слово.

Если указатель слова задается без указания события, в качестве события используется предыдущая команда.

22.1.3 Модификаторы

После необязательного указателя слова, вы можете добавить последовательность из одного или более следующих модификаторов, перед каждым из которых ставится двоеточие.

`h` Удалить заключительную компоненту имени пути, оставляя только начальную.

`r` Удалить заключительный суффикс, имеющий форму '`.`*суффикс*', оставляя базовое имя.

`e` Удалить все, кроме заключительного суффикса.

`p` Напечатать новую команду, но не выполнять ее.

`s/`*старое*`/`*новое*`/`
Заменить первое появление *старое* на *новое* в строке события. На месте '`/`' может использоваться любой разделитель. Разделитель может быть включен в *старое* или *новое* с помощью обратной косой черты. Если в *новое* появляется символ '`&`', он заменяется на *старое*. Одиночная обратная косая черта экранирует '`&`'. Заключительный разделитель необязателен, если он является последним символов во входной строке.

`&` Повторить предыдущую подстановку.

`g` Велит применить изменения ко всей строке события. Используется вместе с '`s`', как в `gs/`*старое*`/`*новое*`/`, или с '`&`'.

Приложение A Форматирование документации

Выпуск 4 GDB содержит уже отформатированную справочную карточку, готовую к печати с PostScript или Ghostscript, в подкаталоге ‘`gdb`’ главного исходного каталога[1]. Если вы можете использовать PostScript или Ghostscript с вашим принтером, вы можете распечатать справочную карточку немедленно из ‘`refcard.ps`’.

Выпуск также включает исходный текст справочной карточки. Вы можете отформатировать ее при помощи TEX, набрав:

```
make refcard.dvi
```

Справочная карточка GDB разработана для печати в режиме *landscape* на бумаге американского размера “letter”; то есть на листе шириной 11 и высотой 8.5 дюймов. Вы должны будете указать этот формат печати в качестве ключа к вашей программе вывода DVI.

Вся документация для GDB поставляется как часть машинно-считываемого дистрибутива. Документация написана в формате Texinfo, который является системой построения документации, использующей один исходный файл, для создания как интерактивного, так и печатного руководства. Вы можете использовать одну из команд форматирования Info, чтобы создать интерактивную версию документации, и TEX (или `texi2roff`) для создания печатной версии.

GDB включает уже отформатированную копию интерактивной версии Info этого руководства в подкаталоге ‘`gdb`’. Основной файл Info— ‘`gdb-5.0/gdb/gdb.info`’, он ссылается на подчиненные файлы ‘`gdb.info*`’ в том же каталоге. В случае необходимости, вы можете распечатать эти файлы, или прочитать их в любом редакторе; но их легче прочитать, используя подсистему `info` в GNU Emacs или автономную программу `info`, доступную как часть дистрибутива GNU Texinfo.

Если вы хотите форматировать эти Info-файлы самостоятельно, вам нужна одна из программ форматирования Info, например `texinfo-format-buffer` или `makeinfo`.

Если у вас установлена программа `makeinfo`, и вы находитесь на верхнем уровне иерархии каталогов GDB (‘`gdb-5.0`’ в случае версии 5.0), вы можете создать файл Info, набрав:

```
cd gdb
make gdb.info
```

Если вы хотите сформировать и распечатать копии этого руководства, вам нужен TEX, программа печати его выходных DVI-файлов и файл определений Texinfo ‘`texinfo.tex`’.

TEX—это программа форматирования; она не печатает файлы непосредственно, а создает выходные файлы, называемые DVI-файлами. Чтобы напечатать сформатированный документ, вам нужна программа печати DVI-файлов. Если в вашей системе установлен TEX, в ней, возможно, есть такая программа. Какую точно команду нужно использовать зависит от вашей системы; *`lpr -d`* является общей; другая программа (для PostScript-устройств)—это *`dvips`*. Команда печати DVI-файлов может требовать имя файла без расширения или с расширением ‘`.dvi`’.

TEX также требует файл макроопределений ‘`texinfo.tex`’. Этот файл сообщает TEX, как форматировать документ, написанный в формате Texinfo. Сам по себе TEX не может читать или форматировать файл Texinfo. ‘`texinfo.tex`’ распространяется с GDB и размещается в каталоге ‘`gdb-`*номер-версии*`/texinfo`’.

Если у вас установлены TEX и программа печати DVI, вы можете отформатировать и распечатать это руководство. Сначала перейдите в подкаталог ‘`gdb`’ главного исходного каталога (например, в ‘`gdb-5.0/gdb`’), и наберите:

```
make gdb.dvi
```

Это передаст ‘`gdb.dvi`’ вашей программе печати DVI.

[1] В ‘`gdb-5.0/gdb/refcard.ps`’ в версии 5.0 выпуска.

Приложение B Установка GDB

GDB поставляется вместе со сценарием `configure`, который автоматизирует процесс подготовки GDB к установке; затем вы можете использовать `make` для построения программы `gdb`.[1]

Дистрибутив GDB включает весь исходный код, который вам понадобится для GDB, в одном каталоге, имя которого обычно составляется добавлением номера версии к '`gdb`'.

Например, дистрибутив GDB версии 5.0 находится в каталоге '`gdb-5.0`'. Этот каталог содержит:

`gdb-5.0/configure` (и файлы поддержки)
: сценарий для конфигурации GDB и всех поддерживаемых библиотек

`gdb-5.0/gdb`
: исходные тексты, специфичные для самого GDB

`gdb-5.0/bfd`
: исходные тексты для библиотеки описания двоичных файлов (Binary File Descriptor)

`gdb-5.0/include`
: включаемые файлы GNU

`gdb-5.0/libiberty`
: исходные тексты для '`-liberty`' библиотеки свободного программного обеспечения

`gdb-5.0/opcodes`
: исходные тексты библиотеки таблиц кодов операций и дисассемблеров

`gdb-5.0/readline`
: исходные тексты интерфейса командной строки GNU

`gdb-5.0/glob`
: исходные тексты подпрограммы GNU сопоставления с образцом имени файла

`gdb-5.0/mmalloc`
: исходные тексты пакета GNU для выделения памяти

Простейший способ сконфигурировать и собрать GDB состоит в выполнении `configure` из исходного каталога '`gdb-`*номер-версии*', который в этом примере есть '`gdb-5.0`'.

Сперва перейдите в исходный каталог '`gdb-`*номер-версии*', если вы еще не находитесь в нем; затем запустите `configure`. Передайте в качестве аргумента идентификатор платформы, на которой будет выполняться GDB.

Например:

```
cd gdb-5.0
./configure платформа
make
```

где *платформа*—идентификатор, такой как '`sun4`' или '`decstation`', задающий платформу, на которой будет выполняться GDB. (Часто вы можете опустить *платформу*; `configure` пытается определить корректное значение, изучая вашу систему.)

Выполнение '`configure` *платформа*' и затем `make` строят библиотеки '`bfd`', '`readline`', '`mmalloc`' и '`libiberty`', и затем сам `gdb`. Сконфигурированные исходные файлы, а также двоичные файлы, остаются в соответствующих исходных каталогах.

[1] Если у вас более новая версия GDB, чем 5.0, просмотрите файл '`README`' в исходном каталоге; мы могли усовершенствовать процедуру установки с момента публикации этого руководства.

`configure` является сценарием оболочки Bourne (`/bin/sh`); если ваша система не распознает это автоматически, когда вы находитесь в другой оболочке, вам может потребоваться выполнить `sh` явно:

```
sh configure платформа
```

Если вы выполните `configure` из каталога, содержащего исходные каталоги для нескольких библиотек или программ, например '`gdb-5.0`' для версии 5.0, `configure` создает файлы конфигурации для всех подкаталогов низшего уровня (если вы не велите ему не этого делать ключем '`-norecursion`').

Вы можете выполнить сценарий `configure` из любого подкаталога из поставки GDB, если вы хотите сконфигурировать только этот подкаталог, но убедитесь, что указали путь к нему.

Например, для версии 5.0, чтобы сконфигурировать только подкаталог `bfd`, введите:

```
cd gdb-5.0/bfd
../configure платформа
```

Вы можете установить `gdb` куда угодно; он не имеет никаких жестко заданных путей. Однако, вы должны удостовериться, что ваша оболочка (определяемая переменной среды '`SHELL`') доступна всем для чтения. Помните, что GDB использует оболочку для запуска вашей программы— некоторые системы не позволяют GDB отлаживать дочерние процессы, чьи программы недоступны для чтения.

B.1 Компиляция GDB в другом каталоге

Если вы хотите запускать версии GDB на нескольких рабочих или целевых машинах, вам нужны различные `gdb`, скомпилированные для каждой комбинации рабочей и целевой машины. `configure` разработан так, чтобы облегчить это, позволяя вам создавать каждую конфигурацию в отдельном подкаталоге, а не в исходном каталоге. Если ваша программа `make` поддерживает возможности '`VPATH`' (GNU `make` это делает), вызов `make` в каждом из этих каталогов строит программу `gdb`, определенную там.

Чтобы построить `gdb` в отдельном каталоге, запустите `configure` с ключем '`-srcdir`', для определения, где искать источник. (Вам также нужно определить путь для поиска `configure` из вашего рабочего каталога. Если путь к `configure` совпадает с параметром '`-srcdir`', ключ '`-srcdir`' можно опустить; он подразумевается.)

Например, в версии 5.0, вы можете построить GDB в отдельном каталоге для Sun 4 так:

```
cd gdb-5.0
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-5.0/configure sun4
make
```

Когда `configure` строит конфигурацию, используя удаленный каталог с источниками, он создает дерево для двоичных файлов с той же структурой (и используя те же имена), что и дерево каталогов с исходными текстами. В этом примере, вы бы нашли библиотеку Sun 4 '`libiberty.a`' в каталоге '`gdb-sun4/libiberty`', и сам GDB в '`gdb-sun4/gdb`'.

Одна из распространенных причин построения нескольких конфигураций GDB в отдельных каталогах состоит в том, чтобы конфигурировать GDB для кросс-компиляции (где GDB запускается на одной машине—*рабочей*, в то время как отлаживаемые программы выполняются на другой машине—*целевой*). Вы определяете целевую машину кросс-отладки ключем `configure` '`-target=`*цель*'.

Когда вы выполняете `make` для построения программы или библиотеки, вы должны выполнять ее из сконфигурированного каталога—того каталога, из которого вы вызывали `configure` (или из одного из его подкаталогов).

`Makefile`, который создает `configure` в каждом исходном каталоге, также выполняется рекурсивно. Если вы ввели `make` в каталоге с исходными файлами, например в ‘`gdb-5.0`’ (или в каталоге, сконфигурированном отдельно посредством ‘`-srcdir=`*имя-каталога*`/gdb-5.0`’), вы построите все требуемые библиотеки, и затем GDB.

Когда у вас имеется несколько рабочих или целевых конфигураций в отдельных каталогах, вы можете запустить `make` для них параллельно (например, если они смонтированы по NFS на каждой рабочей машине); они не будут конфликтовать друг с другом.

B.2 Определение имен рабочих и целевых машин

Спецификации, использованные для рабочих и целевых машин в сценарии `configure`, именуются в соответствии со схемой наименования, состоящей из трех частей, но поддерживаются также некоторые короткие предопределенные синонимы. Полная схема наименования кодирует три фрагмента информации по следующему образцу:

архитектура`-`*производитель*`-`*ОС*

Например, вы можете использовать синоним `sun4` как параметр *платформа*, или как значение *цель* в ключе `--target=`*цель*. Эквивалентное полное имя—‘`sparc-sun-sunos4`’.

Сценарий `configure`, сопровождающий GDB, не предоставляет никаких средств для запроса вывода всех поддерживаемых имен рабочих и целевых машин или их сокращений. `configure` вызывает скрипт оболочки Bourne `config.sub` для отображения сокращений в полные имена; при желании, вы можете посмотреть сценарий или использовать его для проверки ваших предположений о сокращениях. Например:

```
% sh config.sub i386-linux
i386-pc-linux-gnu
% sh config.sub alpha-linux
alpha-unknown-linux-gnu
% sh config.sub hp9k700
hppa1.1-hp-hpux
% sh config.sub sun4
sparc-sun-sunos4.1.1
% sh config.sub sun3
m68k-sun-sunos4.1.1
% sh config.sub i986v
Invalid configuration ‘i986v’: machine ‘i986v’ not recognized
```

`config.sub` также распространяется в исходном каталоге GDB (‘`gdb-5.0`’, для версии 5.0).

B.3 Ключи `configure`

Здесь приводится обзор ключей и параметров `configure`, которые наиболее часто используются для построения GDB. `configure` также имеет несколько других ключей, не представленных здесь. См. Info файл ‘`configure.info`’, node ‘`What Configure Does`’, для полного объяснения `configure`.

```
configure [-help]
          [-prefix=каталог]
          [-exec-prefix=каталог]
          [-srcdir=имя-каталога]
          [-norecursion] [-rm]
          [-target=цель]
          платформа
```

Если хотите, можете вводить ключи с одним ‘`-`’, а не с ‘`--`’; но вы можете сокращать имена ключей, если используете ‘`-`’.

`-help` Отображает быстрый обзор, как вызывать `configure`.

`-prefix=`*каталог*
: Конфигурировать источник, чтобы устанавливать программы и файлы в подкаталогах '*каталога*'.

`-exec-prefix=`*каталог*
: Конфигурировать источник, чтобы устанавливать программы в каталог '*каталог*'.

`-srcdir=`*имя-каталога*
: **Предупреждение: использование этого ключа требует** `GNU make` **или другой программы** `make`**, реализующей возможности** `VPATH`**.**
Используйте этот ключ для создания конфигураций в каталогах, отдельных от исходного каталога GDB. Кроме всего прочего, вы можете использовать его для построения (или поддержки) нескольких конфигураций одновременно в отдельных каталогах. `configure` записывает файлы, относящиеся к конфигурации, в текущий каталог, но принимает меры, чтобы можно было использовать источники в каталоге *имя-каталога*. `configure` создает каталоги внутри рабочего каталога параллельно с исходными каталогами внутри *имя-каталога*.

`-norecursion`
: Конфигурировать только тот уровень каталогов, где выполняется `configure`; не распространять конфигурацию на подкаталоги.

`-target=`*цель*
: Конфигурировать GDB для кросс-отладки программ, выполняемых на указанной *цели*. Без этого ключа GDB конфигурируется для отладки программ, выполняемых на той же машине (*платформе*), что и сам GDB.

  Нет никакого удобного способа сгенерировать список всех доступных целей.

платформа `...`
: Конфигурировать GDB для выполнения на указанной *платформе*.

  Нет никакого удобного способа сгенерировать список всех допустимых платформ.

Существует также много других ключей, но обычно они требуются только для специальных целей.

Алфавитный указатель

default

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

A

B

Г

Д

Ж

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Ц

Ч

Я