

# Debugging with GDB

---

The GNU Source-Level Debugger

Eighth Edition, for GDB version 5.0  
March 2000

Richard Stallman, Roland Pesch, Stan Shebs, et.al.

---

(Send bugs and comments on GDB to [bug-gdb@gnu.org](mailto:bug-gdb@gnu.org).)

*Debugging with GDB*

TeXinfo 1999-03-15.17

Copyright © 1988-2000 Free Software Foundation, Inc.

Published by the Free Software Foundation

59 Temple Place - Suite 330,

Boston, MA 02111-1307 USA

ISBN 1-882114-77-9

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

## GDB の要約

GDB のようなデバッガの目的は、実行中のプログラムの内部において何が起きているのか、あるいは、プログラムがクラッシュしたときに何をしていたのかを知ることができるようにすることにあります。

GDB は、実際のバグを発見できるようにするために主に 4 つのこと（さらに、これらを支援するために他のことも）を行います。

- ユーザ・プログラムを、その動作に影響を与える可能性のあるさまざまなことを指定して起動する
- 指定された条件が成立したときにユーザ・プログラムを停止する
- 停止したときにユーザ・プログラムが何を行っていたかを調べる
- ユーザ・プログラムの内部を変更することによって、1 つのバグの影響を試験的に修正して、ほかのバグについて調べる

GDB を使って C および C++ で記述されたプログラムをデバッグすることができます。詳細については、セクション 9.4 [サポートされる言語], ページ 84 を参照してください。また、セクション 9.4.1 [C and C++], ページ 84 も参照してください。

Modula-2 と Chill のサポートはまだ部分的なものです。Modula-2 に関する情報については、セクション 9.4.2 [Modula-2], ページ 90 を参照してください。Chill に関する情報については、セクション 9.4.3 [Chill], ページ 94 を参照してください。

集合、サブ範囲 (subrange)、ファイル変数、入れ子関数を使っている Pascal プログラムをデバッグすることは、現時点ではできません。Pascal の構文を使って、式の入力、変数の値の表示、およびそれに類することを実行することを、GDB はサポートしていません。

GDB は、Fortran で記述されたプログラムのデバッグに使うことができます。しかし、変数によっては、末尾にアンダースコアを付けて参照する必要がある場合があります。

### フリー・ソフトウェア

GDB はフリー・ソフトウェアであり、GNU General Public License (GPL) により保護されています。あなたは、GPL によって、ライセンスされたプログラムをコピーしたり改造したりする自由を与えられます。しかし、コピーを入手した人は誰でも、そのコピーとともに、そのコピーを修正する自由を手に入れます（つまりソース・コードを入手することができなければならないということです）。また、さらにそのコピーを配布する自由も手に入れます。通常、ソフトウェア会社は、著作権によりユーザの自由を妨げます。Free Software Foundation は GPL を使ってこれらの自由を保護します。

基本的には、GPL は、「あなたはこれらの自由を与えられるが、これらの自由をほかの誰からも奪うことはできない」と主張するライセンスです。

### GDB に貢献した人々

Richard Stallman は、GDB、および、その他の多くの GNU プログラムの最初の開発者です。ほかにも多くの人々が GDB の開発に貢献してきました。この節では、主要な貢献者を紹介したいと思います。フリー・ソフトウェアの素晴らしい点の 1 つは、誰もがそれに貢献する自由があるということです。残念ながら、ここですべての人を紹介することはできません。GDB ディストリビューションに含まれる ‘ChangeLog’ というファイルにおおまかな紹介を載せてあります。

バージョン 2.0 よりもずっと前の変更内容は、いつのまにか紛失してしまいました。

お願い：このセクションへの追加は大歓迎です。あなたやあなたの友人（公平を期するため、あなたの敵も加えておきましょう）が不当にもこのリストから除外されているのであれば、喜んで名前を付け加えます。

彼らの多大な労働が感謝されていないと思われぬように、最初に、GDB の主要なリリースを通じて GDB の面倒を見てきた人々に特に感謝します。その人々とは、Andrew Cagney（リリース 5.0）、Jim Blandy（リリース 4.18）、Jason Molenda（リリース 4.17）、Stan Shebs（リリース 4.14）、Fred Fish（リリース 4.16, 4.15, 4.13, 4.12, 4.11, 4.10, 4.9）、Stu Grossman と John Gilmore（リリース 4.8, 4.7, 4.6, 4.5, 4.4）、John Gilmore（リリース 4.3, 4.2, 4.1, 4.0, 3.9）、Jim Kingdon（リリース 3.5, 3.4, 3.3）、Randy Smith（リリース 3.2, 3.1, 3.0）です。

Richard Stallman は、さまざまな機会に Peter TerMaat、Chris Hanson、Richard Mlynarik の支援を受けながら、2.8 までのリリースを担当しました。

Michael Tiemann は、GDB における GNU C++ サポートのほとんどを開発してくれました。C++ のサポートについては、Per Bothner から重要な貢献がありました。James Clark は GNU C++ のデマングラ（demangler）を開発してくれました。C++ についての初期の仕事は Peter TerMaat によるものです（彼はまた、リリース 3.0 までの一般的なアップデート作業の多くを担当してくれました）。

GDB 4 は、複数のオブジェクト・ファイル・フォーマットを調べるのに、BFD サブルーチン・ライブラリを使用しています。BFD は、David V. Henkel-Wallace、Rich Pixley、Steve Chamberlain、John Gilmore による共同プロジェクトです。

David Johnson は、最初の COFF サポートを開発してくれました。Pace Willison は最初のカプセル化された COFF（encapsulated COFF）のサポートを開発してくれました。

Harris Computer Systems 社の Brent Benson は、DWARF 2 のサポート部分を提供してくれました。

Adam de Boor と Bradley Davis は ISI Optimum V のサポート部分を提供してくれました。Per Bothner、引地信之、Alessandro Forin は、MIPS のサポート部分を提供してくれました。Jean-Daniel Fekete は Sun 386i のサポート部分を提供してくれました。Chris Hanson は HP9000 サポートを改善してくれました。引地信之と長谷井智之は、Sony/News OS 3 のサポート部分を提供してくれました。David Johnson は Encore Umax のサポート部分を提供してくれました。Jyrki Kuoppala は Altos 3068 のサポート部分を提供してくれました。Jeff Law は HP PA と SOM のサポート部分を提供してくれました。Keith Packard は NS32K のサポート部分を提供してくれました。Doug Rabson は Acorn Risc Machine のサポート部分を提供してくれました。Bob Rusk は Harris Nighthawk CX-UX のサポート部分を提供してくれました。Chris Smith は Convex のサポート（および、Fortran デバッグのサポート）部分を提供してくれました。Jonathan Stone は Pyramid のサポート部分を提供してくれました。Michael Tiemann は SPARC のサポート部分を提供してくれました。Tim Tucker は Gould NP1 と Gould Pownode のサポート部分を提供してくれました。Pace Willison は Intel 386 のサポート部分を提供してくれました。Jay Vosburgh は Symmetry のサポート部分を提供してくれました。

Andreas Schwab は M68K Linux のサポート部分を提供してくれました。

Rich Schaefer と Peter Schauer は SunOS 共用ライブラリのサポートを手伝ってくれました。

Jay Fenlason と Roland McGrath は、GDB と GAS がいくつかのマシン命令セットに関して共通の認識を持つようにしてくれました。

Patrick Duval、Ted Goldstein、Vikram Koka、Glenn Engel はリモート・デバッグ機能の開発を手伝ってくれました。Intel 社、Wind River Systems 社、AMD 社、ARM 社はそれぞれ、i960、VxWorks、A29K UDI、RDI ターゲット用のリモート・デバッグ・モジュールを提供してくれました。

Brian Fox は、コマンドライン編集やコマンド履歴を提供する readline ライブラリの開発者です。

SUNY Buffalo の Andrew Beers は言語切り替えのソース・コードと Modula-2 サポート部分を開発し、このマニュアルのプログラミング言語関連 ( Languages ) の章を提供してくれました。

Fred Fish は UNIX System V リリース 4 サポートのほとんどを開発してくれました。彼はまた、C++ のオーバーロードされたシンボルを扱えるようコマンド補完機能を拡張してくれました。

Hitachi America, Ltd. は、H8/300 プロセッサ、H8/500 プロセッサ、および、Super-H プロセッサのサポートを後援してくれました。

NEC は、v850 プロセッサ、Vr4xxx プロセッサ、および、Vr5xxx プロセッサのサポートを後援してくれました。

Mitsubishi ( 三菱 ) は、D10V プロセッサ、D30V プロセッサ、および、M32R/D プロセッサのサポートを後援してくれました。

Toshiba ( 東芝 ) は、TX39 Mips プロセッサのサポートを後援してくれました。

Matsushita ( 松下 ) は、MN10200 プロセッサと MN10300 プロセッサのサポートを後援してくれました。

Fujitsu ( 富士通 ) は、SPARClike プロセッサと FR30 プロセッサのサポートを後援してくれました。

Kung Hsu、Jeff Law、Rick Sladkey はハードウェア・ウォッチポイントのサポートを追加してくれました。

Michael Snyder はトレースポイントのサポートを追加してくれました。

Stu Grossman は gdbserver を開発してくれました。

Jim Kingdon、Peter Schauer、Ian Taylor、Stu Grossman は GDB 全体にわたって、ほとんど数えることができないほどのバグ・フィックスとソース・コードの整理を行ってくれました。

Hewlett-Packard 社の Ben Krepp、Richard Title、John Bishop、Susan Macchia、Kathy Mann、Satish Pai、India Paul、Steve Rehrauer、Elena Zannoni は、PA-RISC 2.0 アーキテクチャ、HP-UX 10.20、10.30、11.0(narrow mode)、HP によるカーネル・スレッドの実装、HP aC++ コンパイラ、および、端末ユーザ・インターフェイスの各サポート部分を提供してくれました。また、このマニュアルの中の HP 固有の情報は、Kim Haase により提供されたものです。

Cygnus Solutions 社は、1991 年以降、GDB の保守作業と GDB の多くの開発作業を後援しています。フルタイムで GDB に関わる仕事をした Cygnus のエンジニアは、Mark Alexander、Jim Blandy、Per Bothner、Kevin Buettner、Edith Epstein、Chris Faylor、Fred Fish、Martin Hunt、Jim Ingham、John Gilmore、Stu Grossman、Kung Hsu、Jim Kingdon、John Metzler、Fernando Nasser、Geoffrey Noer、Dawn Perchik、Rich Pixley、Zdenek Radouch、Keith Seitz、Stan Shebs、David Taylor、Elena Zannoni です。さらに、Dave Brolley、Ian Carmichael、Steve Chamberlain、Nick Clifton、JT Conklin、Stan Cox、DJ Delorie、Ulrich Drepper、Frank Eigler、Doug Evans、Sean Fagan、David Henkel-Wallace、Richard Henderson、Jeff Holcomb、Jeff Law、Jim Lemke、Tom Lord、Bob Manson、Michael Meissner、Jason Merrill、Catherine Moore、Drew Moseley、Ken Raeburn、Gavin Romig-Koch、Rob Savoye、Jamie Smith、Mike Stump、Ian Taylor、Angela Thomas、Michael Tiemann、Tom Tromey、Ron Unrau、Jim Wilson、David Zuhn は、大小様々な貢献をしてくれました。



## 1 GDB セッションのサンプル

その気になれば、このマニュアルを使って GDB のすべてを学習することももちろん可能ですが、GDB を使い始めるには、いくつかのコマンドを知っていれば十分です。本章では、そのようなコマンドについて説明します。

GDB の出力情報との区別が容易につくように、このサンプル・セッションでは、ユーザの入力を `input` のように太字で表わします。

汎用的なマクロ・プロセッサである GNU m4 には、かつて、まだ正式なバージョンがリリースされる以前に、次のような不具合がありました。引用を表わす文字列をデフォルトとは異なるものに変更すると、あるマクロ定義の内部に入れ子状態になっている他のマクロ定義を取り出すために使われるコマンドが、正しく動作しなくなることがある、という不具合です。以下の短い m4 セッションでは、0000 に展開されるマクロ `foo` を定義しています。さらに、m4 の組み込みコマンド `defn` を使って、マクロ `bar` に同一の定義を与えています。ところが、引用の開始文字列を `<QUOTE>` に、引用の終了文字列を `<UNQUOTE>` にそれぞれ変更すると、全く同一の手順で新しい同義語 `baz` を定義しようとしても、うまくいかないのです。

```
$ 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
```

ここで GDB を使って、何が起きているのか調べてみましょう。

```
$ 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 は、必要なときに他のシンボルを見つけるのに最低限必要となるシンボル情報しか読み込みません。その結果、最初のプロンプトが表示されるまでの時間が極めて短いのです。ここで、出力情報がこのマニュアルの紙幅に収まるようにするために、GDB に対して表示幅を通常よりも狭くするよう指示を出してみましょう。

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

m4の組み込みコマンドである `changequote` がどのように動作するのかを調べてみる必要があります。ソースを見ると、関連するサブルーチンが `m4_changequote` であることが分かります。そこで、GDBの `break` コマンドでブレイクポイントを設定してみます。

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

`run` コマンドを使って、GDBの管理下で m4 を走らせます。m4\_changequote サブルーチンに到達するまでは、プログラムは通常どおりの動作をします。

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

```
foo
0000
```

ブレイクポイントでプログラムを停止させるために `changequote` を実行すると、GDBは m4の実行を停止し、停止した箇所のコンテキスト情報を表示します。

```
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))
```

次に `n (next)` コマンドを実行すると、現在停止している関数の中で 1 行だけ処理が実行されます。

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

`set_quotes` というのは、いわくありげなサブルーチンです。`next` コマンドの代わりに `s (step)` コマンドを使うことで、このサブルーチンの中に入ることができます。`step` コマンドは、それがどのサブルーチンの中にあるかということにかかわりなく、次の 1 行に移動します。この場合、次の 1 行は `set_quotes` の中ですから、そこへ移動することになります。

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

m4がその中で現在停止しているサブルーチン（および、そのサブルーチンへの引数）が表示されています。これをスタック・フレーム表示と呼びます。それは、スタックの状態を要約した情報を表示しています。`backtrace` コマンド（あるいは、`bt` と省略することもできます）を使って、現在、スタック全体の中のどこにいるかを知ることができます。`backtrace` コマンドは、個々のアクティブなサブルーチンのスタック・フレームを表示します。

```
(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
```

次に、2、3 行先に進んで、何が起きているのかを見てみましょう。最初の 2 回は、‘s’コマンドを使います。続く 2 回は、xstrdup サブルーチンの中に入ってしまうのを防ぐために、n コマンドを使います。

```
(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);
```

最後に表示された行は、少し妙な感じがします。2 つの変数 lquote、rquote を調べて、本当にそれが、新たに指定された引用開始文字列、引用終了文字列であるかどうか確認することができます。値を調べるには p ( print ) コマンドを使用します。

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

lquote と rquote は確かに引用開始文字列、引用終了文字列のようです。前後関係を調べるには、l ( list ) コマンドを使って、現在停止している行を中心にその前後 10 行を表示します。

```
(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
```

len\_lquote と len\_rquote に値を設定している行を実行させてから、それらの値を調べてみましょう。

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

len\_lquote と len\_rquote が、それぞれ lquote と rquote の長さであるとなると、ここに表示されている値は明らかに誤りです。p コマンドを使って、正しい値を設定することができます。p コマン

ドによって任意の式の値を表示することができますが、ここでいう「式」には、サブルーチンの呼び出しや、値の割り当ても含まれます。

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

新しい引用文字列をセットした状態で、m4の組み込みコマンド `defn` を使用しようとするると発生する問題を修正するには、これだけで十分でしょうか？ `c (continue)` コマンドを使えば、m4に処理を継続させて、実際に問題を発生させていた例を実行することができます。

```
(gdb) c
Continuing.
```

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

```
baz
0000
```

今度はうまくいきました。新たにセットされた引用文字列は、デフォルトの引用文字列と同じように機能しました。問題の原因は、プログラム内の2箇所のタイプ・ミスで、長さの設定が正しく行われていないことにあったようです。EOFを入力して、m4を終了させましょう。

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

‘Program exited normally.’というメッセージは、GDBが出力したもので、m4の実行が終了したことを意味しています。GDBの `quit` コマンドで、GDBセッションを終了することができます。

```
(gdb) quit
```

## 2 GDB の起動と終了

本章では、GDB の起動方法、終了方法を説明します。基本は、以下の 2 つです。

- ‘gdb’ と入力して GDB を起動する
- *quit* または *C-d* を入力して GDB を終了する

### 2.1 GDB の起動

*gdb* というプログラムを実行することで、GDB が起動されます。ひとたび起動されると、GDB は終了を指示されるまで、端末からのコマンド入力を受け付けます。

あるいは、最初から GDB のデバッグ環境を指定するために、さまざまな引数やオプションを指定して *gdb* プログラムを実行することもできます。

ここで説明するコマンドライン・オプションは、さまざまな状況に対応するために設計されたものです。環境によっては、ここで説明するオプションのいくつかは、事実上使用できない場合もあります。

GDB の最も基本的な起動方法は、デバッグされる実行プログラムの名前を引数に指定することです。

```
gdb program
```

起動時に、実行プログラム名とともに、コア・ファイルの名前を指定することもできます。

```
gdb program core
```

あるいは、既に実行中のプロセスをデバッグする場合には、そのプロセス ID を第 2 引数に指定することもできます。

```
gdb program 1234
```

ここでは、GDB はプロセス ID 1234 のプロセスにアタッチします（ただし、‘1234’ という名前のファイルが存在しないというのが条件です。GDB は、まずコア・ファイルの存在を確認します）。

このような第 2 引数の利用が可能であるためには、かなり完成されたオペレーティング・システムが必要になります。ボード・コンピュータに接続して、リモート・デバッガとして GDB を使用する場合には、そもそも「プロセス」という概念がないかもしれませんし、多くの場合、コア・ダンプというものもないでしょう。プロセスへのアタッチやコア・ダンプの読み取りが不可能な場合、GDB は警告を出力します。

*gdb* を起動すると、GDB の無保証性を説明する文章が表示されますが、*-silent* オプションを指定することで、これを表示しないようにすることもできます。

```
gdb -silent
```

コマンドライン・オプションを指定することで、GDB の起動方法をさらに制御することができます。GDB 自身に、使用可能なオプションを表示させることができます。

```
gdb -help
```

のように *gdb* プログラムを実行することで、使用可能なオプションがすべて、その使用方法についての簡単な説明付きで表示されます（短縮して、‘*gdb -h*’ という形で実行しても同じ結果が得られます）。

ユーザの指定したすべてのオプションとコマンドライン引数は、順番に処理されます。‘*-x*’ オプションが指定されている場合は特別で、順序の違いに意味がでています。

### 2.1.1 ファイルの選択

起動された GDB は、指定された引数のうちオプション以外のものは、実行ファイル名およびコア・ファイル名（あるいはプロセス ID）であると解釈します。これは、`-se` オプションと `-c` オプションが指定されたのと同じことです（GDB は、対応するオプション・フラグを持たない最初の引数を `-se` オプション付きと同等とみなし、同じく対応するオプション・フラグを持たない第 2 の引数があれば、これを `-c` オプション付きと同等とみなします）。

例えばほとんどの組み込みターゲット用の場合のように、`configure` によってコア・ファイルのサポートを含むように構成されていない場合、GDB は、このような第 2 引数が存在するとメッセージを出力してそれを無視します。

多くのオプションには、完全形と短縮形があります。以下の一覧では、その両方を示します。オプション名は、他のオプションと区別がつけば、最後まで記述しなくても、GDB によって正しく認識されます（オプション名には `-` ではなく `--` を使うことも可能ですが、ここでは一般的な慣例にしたがうこととします）。

- `-symbols file`
- `-s file`      *file* で指定されるファイルからシンボル・テーブルを読み込みます。
- `-exec file`
- `-e file`      可能であれば、*file* で指定されるファイルを、実行ファイルとして使います。また、このファイルを、コア・ダンプとともにデータを解析するために使います。
- `-se file`      *file* で指定されるファイルからシンボル・テーブルを読み込み、かつ、このファイルを実行ファイルとして使います。
- `-core file`
- `-c file`      *file* で指定されるファイルを解析すべきコア・ダンプとして使います。
- `-c number`      *number* で指定されるプロセス ID を持つプロセスに接続します。これは、`attach` コマンドを実行するのと同様です（ただし、コア・ダンプ形式のファイルが *number* で指定される名前前で存在する場合は、そのファイルを読み取るべきコア・ダンプとして指定したことになります）。
- `-command file`
- `-x file`      *file* で指定されるファイル内に記述された GDB コマンドを実行します。セクション 16.3 [コマンド・ファイル], ページ 165 を参照してください。
- `-directory directory`
- `-d directory`      ソース・ファイルを検索するパスに *directory* で指定されるディレクトリを追加します。
- `-m`
- `-mapped`      注意：このオプションは、すべてのシステムでサポートされているわけではない、オペレーティング・システムのある機能に依存しています。  
システム上で、`mmap` システム・コールによるファイルのメモリへのマッピングが使用可能である場合、このオプションを使うことで、プログラムのシンボル情報を再利用可能なファイルとしてカレント・ディレクトリに書き出させることができます。仮にデバッグ中のプログラム名が `/tmp/fred` であるとする、マップされたシンボル・ファイルは `/tmp/fred.syms` となります。この後の GDB デバッグ・セッションは、このファイルの存在を検出し、そこから迅速にシンボル情報をマップします。この場合、実行プログラムからシンボル情報を読み込むことはありません。

‘.syms’ファイルは、GDB が実行されるホスト・マシンに固有のもので、このファイルは、GDB 内部のシンボル・テーブルのイメージをそのまま保存したものです。これを、複数のホスト・プラットフォーム上において、共有することはできません。

-r

-readnow シンボル・ファイル内のシンボル・テーブル全体をただちに読み込みます。デフォルトの動作では、シンボル情報は必要になるたびに徐々に読み込まれます。このオプションを使うと起動までに時間がかかるようになりますが、その後の処理は速くなります。

-mappedオプションと-readnowオプションは、完全なシンボル情報を含む‘.syms’ファイルを作成するために、通常は一緒に指定されます(‘.syms’ファイルに関する情報については、セクション 12.1 [ファイルを指定するコマンド], ページ 109 を参照してください)。後に使用する目的で‘.syms’を作成するだけで、それ以外には何もしないようにするための GDB の単純な起動方法は、以下のとおりです。

```
gdb -batch -nx -mapped -readnow programname
```

### 2.1.2 モードの選択

GDB を様々なモードで実行することが可能です。例えば、batch モードや quiet モードなどがあります。

-nx

-n 初期化ファイルに記述されたコマンドを実行しません(通常、初期化ファイルは‘.gdbinit’という名前です。ただし、PC 上では‘gdb.ini’となります)。通常は、すべてのコマンド・オプションと引数を処理した後に、GDB は初期化ファイル内のコマンドを実行します。セクション 16.3 [コマンド・ファイル], ページ 165 を参照してください。

-quiet

-silent

-q 紹介メッセージおよびコピーライト・メッセージを表示しません。これらのメッセージは、batch モードでも表示されません。

-batch

batch モードで実行されます。‘-x’オプションで指定されたすべてのコマンド・ファイルを処理した後、終了コード 0 で終了します(‘-n’オプションによって禁止されていなければ、初期化ファイル内に記述されているすべてのコマンドも実行されます)。コマンド・ファイルに記述された GDB コマンドの実行中にエラーが発生した場合には、0 以外の終了コードで終了します。

batch モードは GDB をフィルタとして実行する場合に便利です。例えば、あるプログラムを別のコンピュータ上にダウンロードして実行する場合などです。このような使い方の邪魔にならないよう、

```
Program exited normally.
```

というメッセージは、batch モードでは表示されません(通常このメッセージは、GDB の管理下で実行中のプログラムが終了するときに、必ず表示されます)。

-nowindows

-nw

非ウィンドウ・モード。GDB に GUI が組み込まれている場合、このオプションは GDB に対して、コマンドライン・インターフェイスだけを使うよう指示します。GUI が利用可能ではない場合、このオプションは何の効果も持ちません。

- windows
- w      GDB に GUI が組み込まれている場合、このオプションは、可能であれば GUI を使うよう要求します。
- cd *directory*      カレント・ディレクトリではなく、*directory* で指定されたディレクトリを作業ディレクトリとして、GDB を実行します。
- fullname
- f      GNU Emacs が GDB をサブ・プロセスとして実行するとき、このオプションをセットします。このオプションは、スタック・フレームを表示するときには、必ず完全なファイル名と行番号を標準的な認識可能な書式で出力するよう GDB に対して指示するものです（スタック・フレームは、プログラムの実行が停止されたときに必ず表示されます）。認識可能な書式とは、先頭に 2 つの ‘\032’ 文字、続いてコロンの区切られたファイル名、行番号、桁位置、最後に改行、というものです。Emacs-GDB インターフェイス・プログラムは、フレームに対応するソース・コードを表示させる命令として、2 つの ‘\032’ 文字を使用します。
- epoch      Epoch Emacs-GDB インターフェイスは、GDB をサブ・プロセスとして実行するときに、このオプションをセットします。Epoch が式の値を別ウィンドウ上において表示することができるように、GDB が表示ルーチンを変更するよう通知します。
- annotate *level*      このオプションは GDB 内部の註釈レベルをセットします。その効果は ‘set annotate level’ を使うことと同じです。（章 18 [GDB 註釈], ページ 171 参照）。註釈レベルは、GDB がプロンプトとともにどれだけの情報を表示するかを制御するものです。例えば、式の値、ソース行、その他の出力です。レベル 0 が通常のレベルです。レベル 1 は、GDB が GNU Emacs のサブ・プロセスとして実行されるときに使われます。レベル 2 は、GDB を制御するプログラムに適切な最大限の註釈を表示します。
- async      コマンドライン・インターフェイスにおいて非同期イベント・ループを使用します。GDB は、ユーザのキーボード入力などのすべてのイベントを、特別なイベント・ループを使って処理します。これにより GDB は、デバッグされているプロセスの実行と並行して、ユーザ・コマンドを受け付けて処理することができるようになります。<sup>1</sup> したがってユーザは、次のコマンドを入力するのに、制御が GDB に戻るまで待つ必要はありません。（注：バージョン 5.0 においては、この非同期オペレーションのターゲット側の部分がまだ実装されていないため、‘-async’ は完全には機能しません。）  
標準入力が端末装置に接続されている場合、GDB は、‘-noasync’ オプションによって抑止されていない限り、デフォルトで非同期イベント・ループを使います。
- noasync      コマンドライン・インターフェイスにおいて非同期イベント・ループを抑止します。
- baud *bps*
- b *bps*      GDB によってリモート・デバッグ用に使用されるシリアル・インターフェイスの回線速度（ボーレート、あるいは、秒あたりのビット数）を設定します。
- tty *device*
- t *device*      プログラムの標準入力および標準出力として *device* を使用して実行します。

<sup>1</sup> 原注：MS-DOS/MS-Windows 用の DJGPP ツールによってビルドされた GDB は、このオペレーション・モードをサポートしてはいますが、デバッグ対象が実行中はイベント・ループはサスペンドされています。

- interpreter *interp*  
制御プログラムや制御デバイスとのインターフェイスに、*interp* によって指定されるインタープリタを使用します。このオプションは、GDB をバックエンドとして使用しながら GDB と通信するプログラムによってセットされることを意図したものです。例えば、`--interpreter=mi` を指定すると、GDB は *gdbmi* インターフェイスを使用するようになります ( 章 19 [GDB/MI インターフェイス], ページ 179 参照 )。
- write  
実行ファイルとコア・ファイルを読み書き両用モードでオープンします。これは、GDB 内において `'set write on'` コマンドを実行することと同等です ( セクション 11.6 [プログラムへのパッチ適用], ページ 108 参照 )。
- statistics  
このオプションを指定すると GDB は、個々のコマンドの実行を完了してプロンプトに復帰する前に、消費時間とメモリ使用に関する統計情報を表示するようになります。
- version  
このオプションを指定すると GDB は、バージョン番号と無保証性に関する宣言を表示した後に終了します。

## 2.2 GDB の終了

- quit [*expression*]  
q  
GDB を終了するためには、quit コマンド ( 省略形は q ) を使用するか、あるいは、EOF ( 通常は C-d ) を入力します。 *expression* を指定しない場合、GDB は正常終了します。 *expression* が指定された場合、 *expression* の評価結果をエラー・コードとして終了します。

割り込み ( 多くの場合 C-c ) は GDB を終了させません。割り込みは通常、実行中の GDB コマンドを終了させ、GDB のコマンド・レベルに復帰させます。割り込み文字は、いつ入力しても安全です。というのは、割り込みの発生が危険である間は、GDB が割り込みの発生を抑止するからです。

アタッチされたプロセスやデバイスを制御するために GDB を使用していた場合、detach コマンドでそれを解放することができます ( セクション 4.7 [既に実行中のプロセスのデバッグ], ページ 25 参照 )。

## 2.3 シェル・コマンド

デバッグ・セッションの途中でシェル・コマンドを実行する必要がある場合、GDB を終了したり一時停止させたりする必要はありません。shell コマンドを使用することができます。

- shell *command string*  
*command string* で指定されるコマンド文字列を実行するために標準シェルを起動します。SHELL 環境変数が設定されていれば、その値が実行されるべきシェルを決定します。SHELL 環境変数が設定されていなければ、GDB はデフォルト・シェル ( UNIX システムでは /bin/sh、MS-DOS では 'COMMAND.COM'、等 ) を使用します。

開発環境ではしばしば make ユーティリティが必要とされます。GDB 内部で make ユーティリティを使用する場合は、shell コマンドを使用する必要はありません。

- make *make-args*  
*make-args* で指定される引数とともに make プログラムを実行します。これは、`'shell make make-args'` を実行するのと同じことです。





## 3 GDB コマンド

GDB コマンドの名前は、最初の 2、3 文字に省略することができます。ただし、省略されたコマンド名があいまいであってはなりません。さらに、同じ GDB コマンドを連続して使用する場合には、**(RET)** キーを押すだけで十分です。また、**(TAB)** キーを押すことで、途中まで入力されたコマンド名を補完させることができます (複数の補完候補がある場合には、その一覧を表示します)。

### 3.1 コマンドの構文

GDB コマンドは 1 行で入力されます。1 行の長さには上限がありません。行は、コマンド名で始まり、コマンド名によって意味が決まる引数がそれに続きます。例えば、step コマンドは step を実行する回数を引数に取ります。例えば、'step 5' のようになります。step コマンドは引数なしでも実行可能です。コマンドによっては、全く引数を受け付けないものもあります。

GDB コマンド名は省略可能です。ただし、省略された名前があいまいなものではあってはなりません。省略形は、それぞれのコマンドのドキュメント内に記載されています。場合によっては、あいまいな省略形も許されることがあります。例えば、s は、文字 s で始まるコマンドがほかにも存在するにもかかわらず、step コマンドの省略形として特別に定義されています。ある省略形が使用可能か否かは、それを help コマンドへの引数として使用することで判定可能です。

GDB への入力として空行を与える (**(RET)** キーだけを押す) ことは、1 つ前に実行したコマンドを繰り返すということを意味します。ただし、いくつかのコマンド (例えば、run コマンド) は、この方法で実行を繰り返すことはできません。意図に反して再実行してしまうと問題を引き起こす可能性があるため、繰り返し実行してほしくないようなコマンドの場合です。

list コマンドと x コマンドは、**(RET)** キーにより繰り返し実行すると、新たに引数が生成されて実行されるので、前回実行されたときと全く同様の状態で繰り返し実行されるわけではありません。こうすることで、ソース・コードの内容やメモリの内容を容易に調べることができます。

GDB は、別の用途でも **(RET)** キーを使用します。more ユーティリティと同様の方法で、長い出力を分割して表示する場合です (セクション 15.4 [画面サイズ], ページ 159 参照)。このような場合、**(RET)** キーを余分に押してしまうことは往々にしてありえるので、GDB はこのような表示方法を使用しているコマンドについては、**(RET)** キーによる繰り返し実行を行いません。

テキストの中に # 記号があると、そこから行末まではコメントになります。コメントの部分は実行されません。これは、特にコマンド・ファイルの中で便利です (セクション 16.3 [コマンド・ファイル], ページ 165 参照)。

### 3.2 コマンド名の補完

途中まで入力されたコマンド名は、それがあいまいでなければ、GDB が残りの部分を補完してくれます。また、いつでも、コマンド名の補完候補の一覧を表示してくれます。この機能は、GDB コマンド名、GDB サブ・コマンド名、ユーザ・プログラムのシンボル名に対して有効です。

GDB に単語の残りの部分を補完させたい場合には、**(TAB)** キーを押します。補完候補が 1 つしか存在しない場合、GDB は残りの部分を補完し、ユーザがコマンドを (**(RET)** キーを押すことで) 完結させるのを待ちます。例えば、ユーザが以下のように入力したとしましょう。

```
(gdb) info bre (TAB)
```

GDB は 'breakpoints' という単語の残りの部分を補完します。なぜなら、info コマンドのサブ・コマンドのうち、'bre' で始まるのはこの単語だけだからです。

```
(gdb) info breakpoints
```

この時点で、ユーザは`(RET)`キーを押して `info breakpoints` コマンドを実行するか、あるいは `'breakpoints'` コマンドが実行したいコマンドではなかった場合には、バックスペース・キーを押してこれを消去してから、他の文字を入力することができます (最初から `info breakpoints` コマンドを実行するつもりであれば、コマンド名補完機能ではなくコマンド名の省略形を利用して、`'info bre'` と入力した後、ただちに `(RET)` キーを押してもいいでしょう)。

`(TAB)` キーが押されたときに、2 つ以上の補完候補が存在する場合、GDB はベル音を鳴らします。さらにいくつか文字を入力してから補完を再度試みることも可能ですし、単に続けて `(TAB)` キーを押すことも可能です。後者の場合、GDB は補完候補の全一覧を表示します。例えば、`'make_'` で始まる名前を持つサブルーチンにブレイクポイントを設定したいような場合に、`b make_` まで入力して `(TAB)` キーを入力したところベル音が鳴ったとしましょう。ここで続けて `(TAB)` キーを入力すると、プログラム内の `'make_'` で始まるすべての関数名が表示されます。例えば、以下のように入力したとします。

```
(gdb) b make_ (TAB)
```

ここで GDB はベル音を鳴らします。もう一度 `(TAB)` キーを入力すると、以下のように表示されます。

```
make_a_section_from_file      make_envirom
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_
```

補完候補を表示した後、ユーザが続きを入力できるよう、GDB は途中まで入力された文字列 (ここでは `'b make_'`) を再表示します。

最初から補完候補の一覧を表示したいのであれば、`(TAB)` キーを 2 回押す代わりに `M-?` を入力することもできます。ここで、`M-?` というのは `(META)` ? を意味します。これを入力するには、キーボード上に `(META)` シフト・キーとして指定されたキーがあれば、それを押しながら ? を入力します。`(META)` シフト・キーがない場合には、`(ESC)` キーを押した後、? を入力します。

ときには、入力したい文字列が、論理的には『単語』であっても、GDB が通常は単語の一部に含めない括弧のような文字を含む場合があります。このような場合に単語の補完機能を使用するためには、GDB コマンド内において、そのような単語を ' (単一引用符) で囲みます。

このようなことが必要になる可能性が最も高いのは、C++ 関数名を入力するときでしょう。これは、C++ が関数のオーバーローディング (引数の型の違いによって識別される、同一の名前を持つ関数の複数の定義) をサポートしているからです。例えば、関数 `name` にブレイクポイントを設定する場合、それが `int` 型のパラメータを取る `name(int)` なのか、それとも `float` 型のパラメータを取る `name(float)` なのかをはっきりさせる必要があります。このような場合に単語の補完機能を使用するには、単一引用符 ' を関数名の前に入力します。こうすることによって、`(TAB)` キーまたは `M-?` キーが押されて単語補完が要求されたときに、補完候補の決定には通常よりも多くのことを検討する必要があることが GDB に通知されます。

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

場合によっては、名前の補完をするには引用符を使用する必要があるということを、GDB が自分で認識できることもあります。このような場合、ユーザが引用符を入力していなくても、GDB が (可能な限り補完を行いつつ) 引用符を挿入してくれます。

```
(gdb) b bub (TAB)
```

GDB は入力された 1 行を以下のように変更し、ベル音を鳴らします。

```
(gdb) b 'bubble(
```

一般的には、オーバーロードされたシンボルに対して補完が要求された際に引数リストがまだ入力されていないと、GDB は、引用符が必要であると判断します (そして実際に挿入します)。

オーバーロード関数に関する情報については、セクション 9.4.1.3 [C++ expressions], ページ 86 を参照してください。コマンド `set overload-resolution off` を使用すれば、オーバーロードの解決を無効化することができます。これについては、セクション 9.4.1.7 [C++用の GDB 機能], ページ 88 を参照してください。

### 3.3 ヘルプの表示

`help` コマンドを使うことで、GDB コマンドに関するヘルプ情報を GDB 自身に表示させることができます。

`help`

`h` `help` コマンド (省略形は `h`) を引数なしで実行することで、コマンドのクラス名の簡単な一覧を表示させることができます。

```
(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 class` 一般的なクラス名を引数に指定することで、そのクラスに属するコマンドの一覧を表示させることができます。status クラスを指定した場合の表示例を以下に示します。

```
(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 command

helpの引数にコマンド名を指定することで、そのコマンドの使用法に関する簡単な説明が表示されます。

### apropos args

apropos args コマンドは、args によって指定された正規表現にマッチするものを、すべての GDB コマンド、および、そのドキュメントの中から探します。マッチするものはすべて表示されます。例えば、

```
apropos reload
```

を実行すると、以下のように表示されます。

```

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 args

complete args コマンドにコマンド名の先頭の部分を指定すると、コマンド名の補完候補の一覧を表示します。args には、補完されるべきコマンド名の先頭の文字列を指定します。例えば、

```
complete i
```

は、以下のような結果を表示します。

```

if
ignore
info
inspect

```

これは、GNU Emacs での使用を想定したものです。

help コマンドに加えて、GDB の info コマンドおよび show コマンドを使用することで、ユーザ・プログラムの状態や GDB の状態を問い合わせることができます。どちらのコマンドも、多くの観点からの問い合わせをサポートしています。このマニュアルでは、それぞれを適切と思われる箇所で紹介しています。索引の info や show の部分に、それぞれのサブ・コマンドの紹介されているページが示されています。[インデックス], ページ 269 を参照してください。

info このコマンド ( 省略形は i ) は、ユーザ・プログラムの状態を表わす情報を表示するものです。例えば、info args によってユーザ・プログラムに与えられた引数を、info registers によって現在使用中のレジスタの一覧を、info breakpoints によってユーザが設定したブレイクポイントの一覧を、それぞれ表示することができます。help info によって、info コマンドのサブ・コマンドの完全な一覧が表示されます。

- set**        setコマンドによって、ある式の評価結果を環境変数に割り当てることができます。例えば、GDB のプロンプト文字列を\$記号に変更するには、`set prompt $`を実行します。
- show**        infoコマンドとは異なり、showコマンドは GDB 自身の状態を表わす情報を表示するものです。showコマンドで表示可能な状態のほとんどは、対応する set コマンドで変更可能です。例えば、数値の表示に使用する基数は `set radix`コマンドで制御できます。現在どの基数が使用されているかを単に知るためには、`show radix`コマンドを使用します。
- 変更可能なすべてのパラメータとそれらの現在の値を表示するためには、showコマンドを引数なしで実行します。また、`info set`コマンドを使用することもできます。どちらのコマンドも、同じ情報を出力します。

以下に、対応する set コマンドを持たないという意味で例外的である、3 つの show サブ・コマンドを示します。

**show version**

実行中の GDB のバージョンを表示します。GDB に関する障害レポートには、この情報を含める必要があります。もしも異なるバージョンの GDB を複数使用しているのであれば、現在実行している GDB のバージョンをはっきりさせる必要があるかもしれません。GDB のバージョンが上がるにつれ、新しいコマンドが導入され、古いコマンドはサポートされなくなるかもしれません。多くのシステム・ベンダがさまざまなバージョンの GDB を提供していますし、GNU/Linux ディストリビューションにもさまざまなバージョンの GDB が含まれています。このバージョン番号は、GDB の起動の際に表示されるものと同一です。

**show copying**

GDB のコピー作成許可に関する情報が表示されます。

**show warranty**

GNU の『無保証 (NO WARRANTY)』声明文が表示されます。保証付きの GDB を使っている場合は、保証に関する声明文が表示されます。



## 4 GDB 配下でのプログラムの実行

プログラムを GDB 配下で実行するには、コンパイル時にデバッグ情報を生成する必要があります。

ユーザが選択した環境で、必要に応じて引数を指定して、GDB を起動することができます。ネイティブ環境でデバッグを行っているのであれば、プログラムの入力元と出力先をリダイレクトすること、既に実行中のプロセスをデバッグすること、子プロセスを終了させることもできます。

### 4.1 デバッグのためのコンパイル

プログラムを効率的にデバッグするためには、そのプログラムのコンパイル時にデバッグ情報を生成する必要があります。このデバッグ情報はオブジェクト・ファイルに格納されます。この情報は、個々の変数や関数の型、ソース・コード内の行番号と実行形式コードのアドレスとの対応などを含みます。

デバッグ情報の生成を要求するには、コンパイラの実行時に `-g` オプションを指定します。

多くの C コンパイラでは、`-g` オプションと `-O` オプションを同時に指定することができません。このようなコンパイラでは、デバッグ情報付きの最適化された実行ファイルを生成することができません。

GNU の C コンパイラである GCC は、`-O` オプションの有無にかかわらず、`-g` オプションが指定できます。したがって、最適化されたコードをデバッグすることが可能です。プログラムをコンパイルするときには、常に `-g` オプションを指定することをお勧めします。自分のプログラムは正しいと思うかもしれませんが、自分の幸運を信じて疑わないというのは無意味なことです。

`-g -O` オプションを指定してコンパイルされたプログラムをデバッグするときには、最適化がコードを再調整していることを忘れないでください。デバッガは、実際に存在するコードの情報を表示します。実行されるパスがソース・ファイルの記述と一致していなくても、あまり驚かないでください。これは極端な例ですが、定義されているが実際には使われていない変数を、GDB は認識しません。なぜなら、コンパイラの最適化処理により、そのような変数は削除されるからです。

命令スケジューリング機能を持つマシンなどでは、`-g` を指定してコンパイルされたプログラムでは正しく動作することが、`-g -O` を指定してコンパイルされたプログラムでは正しく動作しないということがあります。`-g -O` を指定してコンパイルされたプログラムのデバッグで何かおかしい点があれば、`-g` だけを指定してコンパイルしてみてください。これで問題が解決するようであれば、(再現環境と一緒に) 障害として私たちに報告してください。

古いバージョンの GNU C コンパイラは、デバッグ情報の生成のためのオプションの 1 つとして `-gg` をサポートしていました。現在の GDB はこのフォーマットをサポートしていません。お手元の GNU C コンパイラにこのオプションがあるようであれば、それは使わないでください。

### 4.2 ユーザ・プログラムの起動

run

r      GDB 配下でユーザ・プログラムの実行を開始するには `run` コマンドを使用してください。(VxWorks 以外の環境では) 最初にプログラム名を指定する必要があります。これには、GDB への引数を使用する方法 (章 2 [GDB の起動と終了], ページ 9 参照) と、`file` コマンドまたは `exec-file` コマンドを使用する方法 (セクション 12.1 [ファイル指定するコマンド], ページ 109 参照) とがあります。

プロセスをサポートする環境でプログラムを実行している場合、runコマンドは下位プロセスを生成し、そのプロセスにプログラムを実行させます（プロセスをサポートしていない環境では、runコマンドはプログラムの先頭アドレスにジャンプします）。

プログラムの実行は、上位プロセスから受け取る情報によって影響されます。GDBはこの情報を指定する手段を提供しています。これは、ユーザ・プログラムが起動される前に実行されていなければなりません（ユーザ・プログラムの実行後にその情報を変更することも可能ですが、その変更結果は、次にプログラムを実行したときに初めて有効になります）。この情報は、4つに分類することができます。

**引数** ユーザ・プログラムに与える引数を、runコマンドへの引数として指定します。ターゲット上でシェルが使用可能であれば、引数を表現するのに通常使用する手法（例えば、ワイルドカード拡張や変数置換など）が利用できるよう、シェルを経由して引数を渡します。UNIXシステムでは、SHELL環境変数によって、使用されるシェルを選択することができます。セクション 4.3 [ユーザ・プログラムの引数], ページ 22 を参照してください。

**環境** ユーザ・プログラムは通常、GDBの環境を継承します。GDBのset environmentコマンドとunset environmentコマンドを使用して、ユーザ・プログラムの実行に影響する環境の一部を変更することができます。セクション 4.4 [ユーザ・プログラムの環境], ページ 23 を参照してください。

#### 作業ディレクトリ

ユーザ・プログラムはGDBの作業ディレクトリを継承します。GDBの作業ディレクトリは、GDBのcdコマンドで設定可能です。セクション 4.5 [ユーザ・プログラムの作業ディレクトリ], ページ 24 を参照してください。

#### 標準入力、標準出力

ユーザ・プログラムは通常、GDBが標準入力、標準出力として使用しているのと同じのデバイスを、標準入力、標準出力として使用します。runコマンドのコマンド・ライン上で、標準入力、標準出力をリダイレクトすることも可能です。また、ttyコマンドによって別のデバイスを割り当てることも可能です。セクション 4.6 [ユーザ・プログラムの入出力], ページ 24 を参照してください。

注意：入出力のリダイレクトは機能しますが、デバッグ中のプログラムの出力を、パイプを使用して他のプログラムに渡すことはできません。このようなことをすると、GDBは誤って、別のプログラムのデバッグを開始してしまうでしょう。

runコマンドを実行すると、ユーザ・プログラムはすぐに実行を始めます。プログラムを停止させる方法については、章 5 [停止と継続], ページ 31 を参照してください。プログラムが停止すると、printコマンドまたはcallコマンドを使用して、プログラム内の関数を呼び出すことができます。章 8 [データの検査], ページ 63 を参照してください。

GDBが最後にシンボル情報を読み込んだ後に、シンボル・ファイルの修正タイムスタンプが変更されている場合、GDBはシンボル・テーブルを破棄して、再読み込みを行います。この場合、GDBは、その時点におけるブレイクポイントの設定を保持しようと試みます。

### 4.3 ユーザ・プログラムの引数

ユーザ・プログラムへの引数は、runコマンドへの引数によって指定可能です。それはまずシェルに渡され、ワイルドカードの展開やI/Oのリダイレクトの後、プログラムに渡されます。SHELL環境変数によって、GDBの使用するシェルが指定されます。SHELL環境変数が定義されていないと、GDBはデフォルト・シェル（UNIXでは/bin/sh）を使用します。



UNIX 以外のシステムでは通常、プログラムは GDB によって直接起動されます。I/O のリダイレクトは、適切なシステム・コールを使用して GDB がエミュレートします。またワイルドカード文字は、シェルではなく、プログラムのスタートアップ・コードによって展開されます。

引数を指定せずに `run` コマンドを実行すると、前回 `run` コマンドを実行したときの引数、または、`set args` コマンドでセットされた引数を使用されます。

`set args` ユーザ・プログラムが次に実行されるときに使用される引数を指定します。`set args` が引数なしで実行された場合、`run` コマンドは、ユーザ・プログラムを引数なしで実行します。一度プログラムに引数を指定して実行すると、次にプログラムを引数なしで実行する唯一の方法は、`run` コマンドを実行する前に `set args` コマンドを実行することです。

`show args` ユーザ・プログラムが実行されるときに渡される引数を表示します。

#### 4.4 ユーザ・プログラムの環境

環境とは、環境変数とその値の集合のことです。環境変数は、慣例として、ユーザ名、ユーザのホーム・ディレクトリ、端末タイプ、実行プログラムのサーチ・パスなどを記録します。通常、環境変数はシェル上で設定され、ユーザの実行するすべてのプログラムによって継承されます。デバッグ時には、GDB を終了・再起動せずに環境を変更して、ユーザ・プログラムを実行できると便利でしょう。

##### `path directory`

`directory` で指定されるディレクトリを環境変数 `PATH` (実行ファイルのサーチ・パス) の先頭に追加します。これは、GDB とユーザ・プログラムの両方に対して有効です。空白類、または、システム依存の区切文字 (UNIX では `;`、MS-DOS と MS-Windows では `;`) で区切られた複数のディレクトリを指定することもできます。環境変数 `PATH` の中に既に `directory` が含まれている場合には、`directory` は環境変数 `PATH` の先頭に移動されます。これにより、`directory` はより早く検索されることになります。

文字列 `$cwd` によって、GDB がパスを検索する時点における作業ディレクトリを参照することができます。`.` (ピリオド) を使用すると、`path` コマンドを実行したディレクトリを参照することになります。`directory` 引数に `.` (ピリオド) が含まれていると、GDB はまずそれを (カレント・ディレクトリに) 置き換えてから、`directory` をサーチ・パスに追加します。

##### `show paths`

実行ファイルを検索するパスの一覧 (環境変数 `PATH` の値) を表示します。

##### `show environment [varname]`

ユーザ・プログラム起動時に渡される環境変数 `varname` の値を表示します。`varname` が指定されない場合は、プログラムに渡されるすべての環境変数の名前と値が表示されます。`environment` は `env` に省略可能です。

##### `set environment varname [=value]`

環境変数 `varname` の値として `value` をセットします。値の変更はユーザ・プログラムに対してのみ有効で、GDB に対しては無効です。`value` には任意の文字列が指定可能です。環境変数の値は単なる文字列であり、その解釈はユーザ・プログラムに委ねられています。`value` は必須パラメータではありません。省略された場合には、変数には空文字列がセットされます。

例えば、以下のコマンドは、デバッグ中のプログラムに対して、それが後に実行されるときにユーザ名は `'foo'` であることを通知します (`=` の前後のスペースは見やすくするためのもので、実際には必要ありません)。

```
set env USER = foo
```

```
unset environment varname
```

ユーザ・プログラムに渡される環境から、環境変数 *varname* を削除します。これは、`'set env varname ='`とは異なります。`unset environment`は、環境変数の値として空文字列をセットするのではなく、環境変数そのものを環境から削除します。

注意: UNIX システムでは、GDB は、環境変数 SHELL により指定されるシェル (環境変数 SHELL が設定されていない場合には `/bin/sh`) を使用してプログラムを実行します。SHELL 環境変数の指定するシェルが初期化ファイルを実行するものである場合 (例えば、C-shell の `'.cshrc'`、BASH の `'.bashrc'`)、初期化ファイルの中で設定された環境変数はユーザ・プログラムに影響を与えます。環境変数の設定は、`'.login'`や`'.profile'`のように、ユーザがシステム内に入るときにのみ実行されるファイルに移したほうがよいでしょう。

#### 4.5 ユーザ・プログラムの作業ディレクトリ

`run` コマンドで実行されるユーザ・プログラムは、実行時の GDB の作業ディレクトリを継承します。GDB の作業ディレクトリは、もともと親プロセス (通常はシェル) から継承したのですが、`cd` コマンドによって、GDB の中から新しい作業ディレクトリを指定することができます。

GDB の作業ディレクトリは、GDB によって操作されるファイルを指定するコマンドに対して、デフォルト・ディレクトリとして機能します。セクション 12.1 [ファイルを指定するコマンド]、ページ 109 を参照してください。

```
cd directory
```

GDB の作業ディレクトリを *directory* にします。

```
pwd
```

GDB の作業ディレクトリを表示します。

#### 4.6 ユーザ・プログラムの入出力

GDB 配下で実行されるプログラムは、デフォルトでは、GDB と同一の端末に対して入出力を行います。GDB は、ユーザとのやりとりのために、端末モードを GDB 用に変更します。このとき、ユーザ・プログラムが使用していた端末モードは記録され、ユーザ・プログラムを継続実行すると、そのモードに戻ります。

```
info terminal
```

ユーザ・プログラムが使用している端末モードに関して GDB が記録している情報を表示します。

`run` コマンドにおいてシェルのリダイレクト機能を使用することによって、ユーザ・プログラムの入出力をリダイレクトすることが可能です。例えば、

```
run > outfile
```

はユーザ・プログラムの実行を開始し、その出力をファイル `'outfile'` に書き込みます。

ユーザ・プログラムの入出力先を指定する別の方法に、`tty` コマンドがあります。このコマンドはファイル名を引数として取り、そのファイルを後に実行される `run` コマンドのデフォルトの入出力先とします。このコマンドはまた、後の `run` コマンドにより生成される子プロセスを制御する端末を変更します。例えば、

```
tty /dev/ttyb
```

は、それ以降に実行される `run` コマンドによって起動されるプロセスのデフォルトの入出力先および制御端末を `/dev/ttyb` 端末とします。

`run` コマンド実行時に明示的にリダイレクト先を指定することで、`tty` コマンドで指定された入出力装置を変更することができますが、制御端末の設定は変更できません。

`tty` コマンドを使用した場合も、`run` コマンドで入力のリダイレクトした場合も、ユーザ・プログラムの入力元だけが変更されます。これらのコマンドを実行しても、GDB の入力元は、ユーザの使用している端末のままです。

#### 4.7 既に実行中のプロセスのデバッグ

`attach process-id`

GDB の外で起動され、既に実行中のプロセスにアタッチします (`info files` コマンドで、現在デバッグ対象となっているプログラムの情報が表示されます)。このコマンドは、プロセス ID を引数に取ります。UNIX プロセスのプロセス ID を知るのに通常使用する方法は、`ps` ユーティリティ、または、シェル・コマンドの `'jobs -l'` の実行です。

`attach` コマンドを実行後 `(RET)` キーを押しても、コマンドは再実行されません。

`attach` コマンドを使用するには、プロセスをサポートする環境でユーザ・プログラムを実行する必要があります。例えば、オペレーティング・システムの存在しないボード・コンピュータのような環境で動作するプログラムに対して、`attach` コマンドを使うことはできません。さらに、ユーザは、プロセスに対してシグナルを送信する権利を持っている必要があります。

`attach` コマンドを使用すると、デバッグは、まずカレントな作業ディレクトリの中で、プロセスにより実行されているプログラムを見つけようとしています。(プログラムが見つからなければ) 次に、ソース・ファイルのサーチ・パス (セクション 7.3 [ソース・ディレクトリの指定], ページ 59 参照) を使用して、プログラムを見つけようとしています。`file` コマンドを使用して、プログラムをロードすることも可能です。セクション 12.1 [ファイル指定するコマンド], ページ 109 を参照してください。

指定されたプロセスをデバッグする準備が整った後に、GDB が最初にするのは、そのプロセスを停止することです。`run` コマンドを使用してプロセスを起動した場合は、通常使用可能なすべての GDB コマンドを使用して、アタッチされたプロセスの状態を調べたり変更したりすることができます。ブレイクポイントの設定、ステップ実行、継続実行、記憶域の内容の変更が可能です。プロセスの実行を継続したいのであれば、GDB がプロセスにアタッチした後に、`continue` コマンドを使用することができます。

`detach`      アタッチされたプロセスのデバッグが終了した場合には、`detach` コマンドを使用してそのプロセスを GDB の管理から解放することができます。プロセスからディタッチしても、そのプロセスは実行を継続します。`detach` コマンド実行後は、ディタッチされたプロセスと GDB は互いに完全に依存関係がなくなり、`attach` コマンドによる別のプロセスへのアタッチや、`run` コマンドによる別のプロセスの起動が可能になります。`detach` コマンドを実行後 `(RET)` キーを押しても、`detach` コマンドは再実行されません。

プロセスがアタッチされている状態で、GDB を終了したり `run` コマンドを使用したりすると、アタッチされたプロセスを終了させてしまいます。デフォルトの状態では、このようなことを実行しようとする、GDB が確認を求めてきます。この確認処理を行うか否かは、`set confirm` コマンドで制御可能です (セクション 15.6 [オプションの警告およびメッセージ], ページ 160 参照)。

## 4.8 子プロセスの終了

**kill**            GDB 配下で実行しているユーザ・プログラムのプロセスを終了させます。

このコマンドは、実行中のプロセスではなく、コア・ダンプをデバッグしたいときに便利です。GDB は、ユーザ・プログラムの実行中は、コア・ダンプ・ファイルを無視します。

いくつかのオペレーティング・システム上では、GDB の管理下でブレイクポイントを設定されている状態のプログラムを、GDB の外で実行することができません。このような場合、kill コマンドを使用することで、デバッガの外でのプログラムの実行が可能になります。

kill コマンドは、プログラムを再コンパイル、再リンクしたい場合にも便利です。というのは、多くのシステムでは、プロセスとして実行中の実行ファイルを更新することはできないからです。次に run コマンドを実行したときに、GDB は、実行ファイルが変更されていることを認識し、シンボル・テーブルを再度読み込みます (この際、その時点でのブレイクポイントの設定を維持しようと試みます)。

## 4.9 マルチスレッド・プログラムのデバッグ

HP-UX や Solaris のようなオペレーティング・システムにおいては、1 つのプログラムが複数のスレッドを実行することができます。「スレッド」の正確な意味は、オペレーティング・システムによって異なります。しかし、一般的には、1 つのアドレス空間を共有するという点を除けば、プログラム内のマルチスレッドは、マルチプロセスと類似しています (アドレス空間の共有とは、複数のスレッドが同一の変数の値を参照したり変更したりすることが可能であるということです)。その一方で、個々のスレッドは自分用のレジスタ、実行スタック、そしておそらくはプライベート・メモリを持ちます。

GDB は、マルチスレッド・プログラムのデバッグ用に、以下のような便利な機能を提供しています。

- 新規スレッド生成の自動的な通知
- スレッドを切り替えるコマンド `'thread threadno'`
- 既存のスレッドに関する情報を問い合わせるコマンド `'info threads'`
- 1 つのコマンドを複数のスレッドに対して実行するコマンド `'thread apply [threadno] [all] args'`
- スレッド固有のブレイクポイント

注意: これらの機能は、スレッドをサポートするオペレーティング・システムをターゲットとして configure によって構成されたすべての GDB で使用可能なわけではありません。GDB がスレッドをサポートしていない環境では、これらのコマンドは無効です。例えば、スレッドをサポートしていないシステム上で GDB の `'info threads'` コマンドを実行しても何も表示されませんし、thread コマンドの実行は常に拒絶されます。

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

GDB のスレッド・デバッグ機能により、ユーザ・プログラムの実行中に、すべてのスレッドを観察することができます。ただし、GDB に制御権のある状態では、特定の 1 つのスレッドだけがデバッグの対象となります。このスレッドは、カレント・スレッドと呼ばれます。デバッグ用のコマンドは、カレント・スレッドの立場から見たプログラムの情報を表示します。

ユーザ・プログラム内部において新しいスレッドの存在を検出すると、GDB は、‘[New *systag*]’ という形式で、ターゲット・システム上におけるこのスレッドの ID を表示します。ここで *systag* とはスレッドの ID で、その形式はシステムによって異なります。例えば、LynxOS 上では、GDB が新しいスレッドを検出すると、

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

のように表示されます。一方、SGI のシステム上では、*systag* は単に ‘process 368’ のような形式で、これ以外の情報は含まれません。

GDB は、デバッグを行うために、ユーザ・プログラム内の個々のスレッドに対して整数値のスレッド番号を独自に割り当てます。

info threads

その時点においてユーザ・プログラム中に存在するすべてのスレッドに関する要約を表示します。個々のスレッドに関して、以下の情報が（列挙された順に）表示されます。

1. GDB により割り当てられたスレッド番号
2. ターゲット・システムのスレッド ID ( *systag* )
3. スレッドのカレントなスタック・フレームの要約

GDB により割り当てられたスレッド番号の左のアスタリスク ‘\*’ は、そのスレッドがカレント・スレッドであることを意味しています。

以下に例を示します。

```
(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
```

HP-UX システムでは以下のようになります。

GDB は、デバッグを行うために、独自のスレッド番号–スレッドの生成順に割り当てられるサイズの小さい整数値–を、ユーザ・プログラムの個々のスレッドに割り当てます。

GDB は、ユーザ・プログラムの中に新しいスレッドの存在を検出するたびに、‘[New *systag*]’ という形式のメッセージで、GDB のスレッド番号とそのスレッドに対するターゲット・システムの識別子を表示します。*systag* はスレッドの ID で、その形式はシステムによって異なります。例えば、HP-UX 上では、GDB が新しいスレッドの存在を検出すると、

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

というメッセージが表示されます。

info threads

その時点においてユーザ・プログラム中に存在するすべてのスレッドに関する要約を表示します。個々のスレッドに関して、以下の情報が（列挙された順に）表示されます。

1. GDB によって割り当てられたスレッド番号
2. ターゲット・システムのスレッド識別子 ( *systag* )
3. そのスレッドのカレントなスタック・フレームの要約情報

GDB により割り当てられたスレッド番号の左側にあるアスタリスク ‘\*’ は、そのスレッドが、カレント・スレッドであることを示しています。

以下に、例を示します。

```
(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 *threadno*

スレッド番号 *threadno* を割り当てられたスレッドをカレント・スレッドとします。このコマンドの引数 *threadno* は、‘info threads’コマンドの出力の最初のフィールドに表示される、GDB 内部のスレッド番号です。GDB は、指定されたスレッドのシステム上の ID とカレントなスタック・フレームの要約を表示します。

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

‘[New ...]’メッセージと同様、‘Switching to’の後ろに表示される情報の形式は、そのシステムにおけるスレッドの識別方法に依存します。

#### thread apply [*threadno*] [*all*] *args*

thread apply コマンドにより、1 つのコマンドを 1 つ以上のスレッドに対して実行することができます。実行対象となるスレッドのスレッド番号を、引数 *threadno* に指定します。*threadno* は、‘info threads’コマンドの出力の最初のフィールドに表示される、GDB 内部のスレッド番号です。すべてのスレッドに対してコマンドを実行するには、thread apply all *args* コマンドを使用してください。

GDB がユーザ・プログラムを停止させるとき、その理由がブレイクポイントであれシグナルの受信であれ、ブレイクポイントに到達したスレッド、または、シグナルを受信したスレッドが自動的に選択されます。GDB は、‘[Switching to *systag*]’という形式のメッセージでそのスレッドを示し、コンテキスト切り替えの発生に注意を促します。

複数スレッドを持つプログラムの停止時や起動時の GDB の動作の詳細については、セクション 5.4 [マルチスレッド・プログラムの停止と起動]、ページ 49 を参照してください。

また、複数スレッドを持つプログラムの中におけるウォッチポイントについては、セクション 5.1.2 [ウォッチポイントの設定]、ページ 36 を参照してください。

## 4.10 マルチプロセス・プログラムのデバッグ

ほとんどのシステムにおいて GDB は、fork 関数を使用して新たにプロセスを生成するプログラムのデバッグに関して特別な機能を提供していません。プログラムが fork を実行するとき、GDB は引き続き親プロセスのデバッグを継続し、子プロセスは妨げられることなく実行されます。子プロセスが実行するコードにブレイクポイントを設定してあると、子プロセスは SIGTRAP シグナルを受信し、(そのシグナルをキャッチする処理がなければ) 子プロセスは終了してしまいます。

しかし、子プロセスをデバッグしたい場合には、それほど困難ではない回避策があります。fork の呼び出し後に子プロセスが実行するソース・コードの中に、sleep 関数の呼び出しを加えてくださ

い。GDB に子プロセスのデバッグをさせる理由がないときに遅延が発生することのないように、特定の環境変数が設定されているときや特定のファイルが存在するときのみ、sleep関数を呼び出すようにするとよいでしょう。子プロセスがsleepを呼び出している間に、psユーティリティを使用して子プロセスのプロセス ID を獲得します。次に、GDB に対して ( 親プロセスもデバッグするのであれば、新たに GDB を起動して、その GDB に対して ) 子プロセスにアタッチするよう指示してください ( セクション 4.7 [既に実行中のプロセスのデバッグ], ページ 25 参照 )。これ以降は、通常の方法でプロセスにアタッチした場合と全く同様に、子プロセスのデバッグが可能です。

これに対して HP-UX ( おそらくバージョン 11.x 以降のみであると思われます ) 上の GDB は、fork関数、または、vfork関数を使用して新たにプロセスを生成するプログラムのデバッグをサポートしています。

デフォルトでは、プログラムが forkを実行するとき、GDB は、引き続き親プロセスのデバッグを継続し、子プロセスは、妨げられることなく実行されます。

親プロセスではなく子プロセスの実行を追跡したい場合は、コマンド set follow-fork-modeを使用します。

set follow-fork-mode *mode*

プログラムによる forkや vforkの呼び出しに反応するよう、デバッガを設定します。forkや vforkの呼び出しは、新しいプロセスを生成します。mode は、以下のいずれかです。

- |        |  |
|--------|--|
| parent | forkの後、元のプロセスをデバッグします。子プロセスは、妨げられることなく実行されます。これがデフォルトです。 |
| child  | forkの後、新しいプロセスをデバッグします。親プロセスは、妨げられることなく実行されます。           |
| ask    | 上記のどちらを選択するかを、デバッガが問い合わせます。                              |

show follow-fork-mode

forkや vforkの呼び出しに対して、現在、デバッガがどのように反応するよう設定されているかを表示します。

子プロセスをデバッグするよう要求されているときに、vforkに続けて execが呼び出されると、GDB は、新しいターゲットを、設定されている最初のブレイクポイントまで実行します。元のプログラムの mainにブレイクポイントがセットされていると、子プロセスの mainにもブレイクポイントがセットされます。

子プロセスが vforkによって生成 ( spawn ) された場合、execの呼び出しが完了するまで、子プロセス、親プロセスのどちらもデバッグすることはできません。

execの呼び出し後に、GDB に対して runコマンドを発行すると、新しいターゲットが再始動されます。親プロセスを再始動するには、親プロセスの実行ファイル名を引数に指定して、fileコマンドを使用します。

catchコマンドを使用することによって、fork、vfork、execの呼び出しが行われるたびに、GDB を停止させることができます。セクション 5.1.3 [キャッチポイントの設定], ページ 38 を参照してください。





## 5 停止と継続

デバッガを使用する主な目的は、プログラムが終了してしまう前に停止させたり、問題のあるプログラムを調査して何が悪いのかを調べたりすることにあります。

GDB 内部においてプログラムが停止する原因はいくつかあります。例えば、シグナルの受信、ブレイクポイントへの到達、stepコマンドのような GDB コマンドの実行後の新しい行への到達などです。プログラムが停止すると、変数の値の調査や設定、新しいブレイクポイントの設定、既存のブレイクポイントの削除などを行った後に、プログラムの実行を継続することができます。通常、GDB が表示するメッセージは、ユーザ・プログラムの状態について多くの情報を提供してくれます。ユーザはいつでも明示的にこれらの情報を要求することができます。

info program

ユーザ・プログラムの状態に関する情報を表示します。表示される情報は、そのプログラムの実行状態（実行中か否か）、そのプログラムのプロセス、プログラムが停止した理由です。

### 5.1 ブレイクポイント、ウォッチポイント、キャッチポイント

ブレイクポイントによって、プログラム内のある特定の箇所に到達するたびに、プログラムを停止することができます。個々のブレイクポイントについて、そのブレイクポイントにおいてプログラムを停止させるためには満足されなければならない、より詳細な条件を設定することができます。ブレイクポイントの設定は、いくつかある break コマンドのいずれかによって行います（セクション 5.1.1 [ブレイクポイントの設定], ページ 32 参照）。行番号、関数名、プログラム内における正確なアドレスを指定することで、プログラムのどこで停止するかを指定することができます。

HP-UX、SunOS 4.x、SVR4、Alpha OSF/1 上では、実行開始前に共用ライブラリ内にブレイクポイントを設定することもできます。HP-UX システムでは、ちょっとした制約があります。プログラムによって直接呼び出されるのではない共用ライブラリ・ルーチン（例えば、pthread\_create の呼び出しにおいて、引数として指定されるルーチン）にブレイクポイントをセットするためには、そのプログラムの実行が開始されるまで待たなければなりません。

ウォッチポイントは、ある式の値が変化したときにユーザ・プログラムを停止させる、特別なブレイクポイントです。ウォッチポイントは、他のブレイクポイントと同じように管理することができますが、設定だけは特別なコマンドで行います（セクション 5.1.2 [ウォッチポイントの設定], ページ 36 参照）。有効化、無効化、および削除を行うときに使用する各コマンドは、対象がブレイクポイントであってもウォッチポイントであっても同一です。

ブレイクポイントで GDB が停止するたびに、常に自動的にユーザ・プログラム内のある値を表示させるようにすることができます。セクション 8.6 [自動表示], ページ 68 を参照してください。

キャッチポイントは、C++ の例外の発生やライブラリのローディングのようなある種のイベントが発生したときに、ユーザ・プログラムを停止させる、また別の特殊なブレイクポイントです。ウォッチポイントと同様、キャッチポイントを設定するために使用する特別なコマンドがあります。（セクション 5.1.3 [キャッチポイントの設定], ページ 38 参照）。しかし、この点を除けば、キャッチポイントを他のブレイクポイントと同様に管理することができます。（ユーザ・プログラムがシグナルを受信したときに停止するようにするためには、handle コマンドを使用します。セクション 5.3 [シグナル], ページ 48 を参照してください）。

ユーザが新規に作成した個々のブレイクポイント、ウォッチポイント、キャッチポイントに対して、GDB は番号を割り当てます。この番号は 1 から始まる連続する整数値です。ブレイクポイントのさ

さまざまな側面を制御するコマンドの多くにおいて、変更を加えたいブレイクポイントを指定するのにこの番号を使用します。個々のブレイクポイントを有効化、無効化することができます。無効化されたブレイクポイントは、再度有効化されるまで、ユーザ・プログラムの実行に影響を与えません。

いくつかの GDB コマンドでは、操作対象となるブレイクポイントの範囲を指定することができます。ブレイクポイントの範囲とは、‘5’のような単一のブレイクポイント番号、または、‘5-7’のように、昇順に並んだ 2 つのブレイクポイント番号をハイフンで区切ったもののいずれかです。あるコマンドに対してブレイクポイントの範囲が指定された場合、その範囲にあるすべてのブレイクポイントが操作の対象となります。

### 5.1.1 ブレイクポイントの設定

ブレイクポイントは、`break` コマンド ( 省略形は `b` ) によって設定されます。デバッガのコンビニエンス変数 `$bpnum` に、最後に設定されたブレイクポイントの番号が記録されます。コンビニエンス変数の用途については、セクション 8.9 [コンビニエンス変数], ページ 76 を参照してください。

ブレイクポイントの設定箇所を指定する方法はいくつかあります。

#### `break function`

関数 *function* のエントリにブレイクポイントを設定します。ソース言語が ( 例えば C++ のように ) シンボルのオーバーロード機能を持つ場合、*function* は、プログラムを停止させる可能性を持つ 1 つ以上の箇所を指すことがあります。このような状況に関する説明については、セクション 5.1.8 [ブレイクポイント・メニュー], ページ 44 を参照してください。

#### `break +offset`

#### `break -offset`

その時点において選択されているスタック・フレームにおいて実行が停止している箇所から、指定された行数だけ先または手前にブレイクポイントを設定します。スタック・フレームについては、セクション 6.1 [スタック・フレーム], ページ 51 を参照してください。

#### `break linenum`

カレントなソース・ファイル内の *linenum* で指定される行番号を持つ行に、ブレイクポイントを設定します。ここで「カレントなソース・ファイル」とは、最後にソース・コードが表示されたファイルを指します。このブレイクポイントは、その行に対応するコードが実行される直前に、ユーザ・プログラムを停止させます。

#### `break filename:linenum`

*filename* で指定されるソース・ファイルの *linenum* で指定される番号の行に、ブレイクポイントを設定します。

#### `break filename:function`

*filename* で指定されるソース・ファイル内の *function* で指定される関数エントリにブレイクポイントを設定します。同じ名前の関数が複数のファイルに存在する場合以外は、ファイル名と関数名を同時に指定する必要はありません。

#### `break *address`

*address* で指定されるアドレスにブレイクポイントを設定します。これは、プログラムの中の、デバッグ情報やソース・ファイルが手に入らない部分にブレイクポイントを設定するのに使用できます。

**break**      引数なしで実行されると、breakコマンドは、選択されたスタック・フレーム内において次に実行される命令にブレイクポイントを設定します（章 6 [スタックの検査], ページ 51 参照）。最下位にあるスタック・フレーム以外のフレームが選択されていると、このブレイクポイントは、制御がそのフレームに戻ってきた時点でユーザ・プログラムを停止させます。これが持つ効果は、選択されたフレームの下位にあるフレームにおいて finish コマンドを実行するのと似ています。ただし、1 つ異なるのは、finish コマンドがアクティブなブレイクポイントを残さないという点です。最下位のスタック・フレームにおいて引数なしで break コマンドを実行した場合、GDB は、そのときに停止していた箇所に次に到達したときにユーザ・プログラムを停止させます。これは、ループの内部では便利でしょう。

GDB は通常、実行を再開したときに、最低でも 1 命令が実行されるまでの間はブレイクポイントの存在を無視します。そうでなければ、ブレイクポイントで停止した後、そのブレイクポイントを無効にしない限り先へ進めないことになってしまいます。この規則は、ユーザ・プログラムが停止したときに既にそのブレイクポイントが存在したか否かにかかわらず、適用されます。

**break ... if cond**

*cond* で指定される条件式付きでブレイクポイントを設定します。このブレイクポイントに達すると、必ず条件式 *cond* が評価されます。その結果がゼロでない場合、すなわち、*cond* の評価結果が真である場合のみ、ユーザ・プログラムを停止します。‘...’の部分には、これまでに説明してきた、停止箇所を指定するための引数のいずれかが入ります（‘...’は省略も可能です）。ブレイクポイントの条件式の詳細については、セクション 5.1.6 [ブレイクポイントの成立条件], ページ 41 を参照してください。

**tbreak args**

プログラムを 1 回だけ停止させるブレイクポイントを設定します。*args* の部分は break コマンドと同様であり、ブレイクポイントも同じように設定されますが、tbreak により設定されたブレイクポイントは、プログラムが最初にそこで停止した後に自動的に削除されます。セクション 5.1.5 [ブレイクポイントの無効化], ページ 40 を参照してください。

**hbreak args**

ハードウェアの持つ機能を利用したブレイクポイントを設定します。*args* の部分は break コマンドと同様であり、ブレイクポイントも同じように設定されますが、hbreak により設定されるブレイクポイントは、ハードウェアによるサポートを必要とします。ターゲット・ハードウェアによっては、このような機能を持たないものもあるでしょう。これの主な目的は、EPROM/ROM コードのデバッグであり、ある命令を変更することなく、その命令にブレイクポイントを設定することです。これは、SPARClike DSU といくつかの x86 ベースのターゲットが提供する、新しいトラップ発生機能と組み合わせて使用することができます。デバッグ・レジスタに割り当てられたデータ・アドレスや命令アドレスをプログラムがアクセスすると、これらのターゲットはトラップを発生させます。ただし、ハードウェアの提供するブレイクポイント・レジスタが取ることのできるブレイクポイントの数には限りがあります。例えば、DSU ではデータ・ブレイクポイントを同時には 2 つまでしか取れないので、3 つ以上使用しようとする、GDB はそれを拒絶します。このような場合、不要になったハードウェア・ブレイクポイントを削除または無効化（セクション 5.1.5 [ブレイクポイントの無効化], ページ 40 参照）してから、新しいハードウェア・ブレイクポイントを設定してください。セクション 5.1.6 [ブレイクポイントの成立条件], ページ 41 を参照してください。

**thbreak args**

ハードウェアの機能を利用して、プログラムを 1 回だけ停止させるブレイクポイントを設定します。*args* の部分は *hbreak* コマンドと同様であり、ブレイクポイントも同じように設定されます。しかし、*tbreak* コマンドの場合と同様、最初にプログラムがそこで停止した後に、このブレイクポイントは自動的に削除されます。また、*hbreak* コマンドの場合と同様、このブレイクポイントはハードウェアによるサポートを必要とするものであり、ターゲット・ハードウェアによっては、そのような機能がないこともあるでしょう。セクション 5.1.5 [ブレイクポイントの無効化], ページ 40 を参照してください。また、セクション 5.1.6 [ブレイクポイントの成立条件], ページ 41 もあわせて参照してください。。

**rbreak regex**

*regex* で指定される正規表現にマッチするすべての関数にブレイクポイントを設定します。このコマンドは、正規表現にマッチしたすべての関数に無条件ブレイクポイントを設定し、設定されたすべてのブレイクポイントの一覧を表示します。設定されたブレイクポイントは、*break* コマンドで設定されたブレイクポイントと同様に扱われます。他のすべてのブレイクポイントと同様の方法で、削除、無効化、および条件の設定が可能です。

正規表現の構文は、*'grep'* のようなツールによって使用される標準的なものです。シェ尔によって使用される構文とは異なることに注意してください。例えば、*foo\** は、*foo* の後ろにゼロ個以上の *o* が続く部分を含む名前を持つすべての関数にマッチします。ユーザの指定する正規表現の先頭と末尾には暗黙のうちに *.\** があるものと想定されますので、名前の先頭が *foo* である関数にだけマッチさせたいときは、*^foo* を使ってください。

C++ プログラムのデバッグにおいて、あるオーバーロードされたメンバ関数が、特別なクラスだけが持つメンバ関数というわけではない場合、そのメンバ関数にブレイクポイントを設定するのに、*rbreak* コマンドは便利です。

**info breakpoints [n]**

**info break [n]**

**info watchpoints [n]**

設定された後、削除されていない、すべてのブレイクポイント、ウォッチポイント、キャッチポイントの一覧を表示します。それぞれについて、以下の情報が表示されます。

ブレイクポイント番号

タイプ      ブレイクポイント、ウォッチポイント、または、キャッチポイント

廃棄          ブレイクポイントに次に到達したときに、無効化または削除されるべくマークされているか否かを示します。

有効 / 無効   有効なブレイクポイントを 'y'、有効でないブレイクポイントを 'n' で示します。

アドレス      ユーザ・プログラム内のブレイクポイントの位置をメモリ・アドレスとして示します。

対象          ユーザ・プログラムのソース内におけるブレイクポイントの位置を、ファイル名および行番号で示します。

ブレイクポイントが条件付きのものである場合、*info break* コマンドは、そのブレイクポイントに関する情報の次の行に、その条件を表示します。ブレイクポイント・コマンドがあれば、続いてそれが表示されます。

`info break` コマンドに引数としてブレイクポイント番号  $n$  が指定されると、その番号に対応するブレイクポイントだけが表示されます。コンビニエンス変数 `$_`、および、`x` コマンドのデフォルトの参照アドレスには、一覧の中で最後に表示されたブレイクポイントのアドレスが設定されます (セクション 8.5 [メモリの調査], ページ 67 参照)。

`info break` コマンドは、ブレイクポイントに到達した回数を表示します。これは、`ignore` コマンドと組み合わせると非常に便利です。まず、`ignore` コマンドによってブレイクポイントへの到達をかなりの回数無視するように設定します。プログラムを実行し、`info break` コマンドの出力結果から何回ブレイクポイントに到達したかを調べます。再度プログラムを実行し、今度は前回の実行時に到達した回数より 1 だけ少ない回数だけ無視するように設定します。こうすることで、前回の実行時にそのブレイクポイントに最後に到達したときと同じ状態でプログラムを停止させることができます。

GDB では、ユーザ・プログラム内の同一箇所に何度でもブレイクポイントを設定することができます。これは、くだらないことでも、無意味なことでもありません。設定されるブレイクポイントが条件付きのものである場合、これはむしろ有用です (セクション 5.1.6 [ブレイクポイントの成立条件], ページ 41 参照)。

GDB 自身が、特別な目的でユーザ・プログラム内部にブレイクポイントを設定することがあります。例えば、(C プログラムにおける) `longjmp` を適切に処理するためなどです。これらの内部的なブレイクポイントには -1 から始まる負の番号が割り当てられます。‘`info breakpoints`’ コマンドは、このようなブレイクポイントを表示しません。

これらのブレイクポイントは、GDB の保守コマンド ‘`maint info breakpoints`’ で表示することができます。

`maint info breakpoints`

‘`info breakpoints`’ コマンドと同様の形式で呼び出され、ユーザが明示的に設定したブレイクポイントと、GDB が内部的な目的で使用しているブレイクポイントの両方を表示します。内部的なブレイクポイントは、負のブレイクポイント番号で示されます。タイプ欄にブレイクポイントの種類が表示されます。

`breakpoint`

明示的に設定された普通のブレイクポイント

`watchpoint`

明示的に設定された普通のウォッチポイント

`longjmp`

`longjmp` が呼び出されたときに正しくステップ処理ができるように、内部的に設定されたブレイクポイント

`longjmp resume`

`longjmp` のターゲットとなる箇所に内部的に設定されたブレイクポイント

`until`

GDB の `until` コマンドで一時的に使用される内部的なブレイクポイント

`finish`

GDB の `finish` コマンドで一時的に使用される内部的なブレイクポイント

`shlib events`

共用ライブラリ・イベント

### 5.1.2 ウォッチポイントの設定

ウォッチポイントを設定することで、ある式の値が変化したときに、プログラムの実行を停止させることができます。その値の変更が、プログラムのどの部分で行われるかをあらかじめ知っている必要はありません。

システムによって、ウォッチポイントがソフトウェアによって実装されていることもあれば、ハードウェアによって実装されていることもあります。GDBは、ユーザ・プログラムをシングル・ステップ実行して、そのたびに変数の値をテストすることによって、ソフトウェア・ウォッチポイントを実現しています。これは、通常の実行と比較すると、何百倍も遅くなります。(それでも、プログラムのどの部分が問題を発生させたのか全く手掛りのない誤りを見つけることができるのであれば、十分価値のあることかもしれません)。

HP-UX、Linux、および、その他のいくつかの x86 ベースのターゲット上では、GDB にハードウェア・ウォッチポイントのサポートも組み込まれています。これを使用すれば、ユーザ・プログラムの実行が遅くなることはありません。

`watch expr`

`expr` で指定される式に対してウォッチポイントを設定します。プログラムがこの式の値を書き換えるときに、GDB はプログラムの実行を停止させます。

`rwatch expr`

`expr` で指定される対象が読み取りアクセスされるときにプログラムを停止させるウォッチポイントを設定します。

`awatch expr`

`expr` で指定される対象が読み取りアクセス、または、書き込みアクセスされるときにプログラムを停止させるウォッチポイントを設定します。

`info watchpoints`

ウォッチポイント、ブレイクポイント、キャッチポイントの一覧を表示します。これは、`info break`と同じです。

GDB は、可能であれば、ハードウェア・ウォッチポイントを設定します。ハードウェア・ウォッチポイントを設定した場合は高速な実行が可能であり、デバッガは、変更を引き起こした命令のところで、値の変更を報告することができます。ハードウェア・ウォッチポイントを設定できない場合、GDB は、ソフトウェア・ウォッチポイントを設定します。これは、実行速度も遅く、値の変更は、その変更が実際に発生した後に、その変更を引き起こした命令のところではなく、1 つ後ろの文のところで報告されます。

`watch` コマンドを実行すると、ハードウェア・ウォッチポイントの設定が可能な場合には、GDB は、以下のような報告を行います。

Hardware watchpoint *num*: *expr*

現在のところ、`awatch` コマンドと `rwatch` コマンドは、ハードウェア・ウォッチポイントしかセットすることができません。これは、監視対象となっている式の値を変更しないようなデータへのアクセスは、すべての命令をその実行時に検査することなくしては実現できないからです。GDB は、現在のところこのようなことを行っていません。`awatch` コマンドや `rwatch` コマンドでハードウェア・ブレイクポイントを設定できないことが判明すると、GDB は以下のようなメッセージを出力します。

Expression cannot be implemented with read/access watchpoint.

ときには、監視対象となる式のデータ型が、ターゲット・マシン上においてハードウェア・ウォッチポイントが処理できる範囲を超えているために、GDB がハードウェア・ウォッチポイントを設定

できないということが起こり得ます。例えば、システムによっては、4 バイトまでの領域しか監視できません。このようなシステムでは、( 典型的には 8 バイトである ) 倍精度浮動小数点数をもたす式に対するハードウェア・ウォッチポイントはセットできません。1 つの回避策として、大きな領域を連続する小さな領域に分割して、それぞれを別のウォッチポイントによって監視するという事は可能であるかもしれません。

ハードウェア・ウォッチポイントをあまりにも数多くセットすると、プログラムの実行を再開したときに、GDB がそのすべてを設定することができない可能性があります。アクティブなウォッチポイントの正確な数は、プログラムの実行が再開されるその瞬間まで分らないため、ユーザがウォッチポイントをセットしている最中には、GDB はこの点について警告を発することができないかもしれません。以下のような警告が、プログラムの実行再開時に初めて表示されることになります。

```
Hardware watchpoint num: Could not insert watchpoint
```

このようなことが発生した場合は、ウォッチポイントをいくつか削除するか、もしくは、無効にしてください。

SPARClike DSU は、デバッグ・レジスタに割り当てられたデータ・アドレスや命令アドレスにプログラムがアクセスすると、トラップを発生させます。データ・アドレスについては、DSU が watch コマンドを支援しています。しかし、ハードウェアの提供するブレイクポイント・レジスタは、データ・ウォッチポイントを 2 つまでしか取れず、その 2 つは同じ種類のウォッチポイントでなければなりません。例えば、2 つのウォッチポイントを、両方とも watch コマンドでセットすること、両方とも rwatch コマンドでセットすること、あるいは、両方とも awatch コマンドでセットすることはいずれも可能ですが、それぞれを異なるコマンドでセットすることはできません。異なる種類のウォッチポイントを同時にセットしようとしても、コマンドの実行を GDB が拒否します。このような場合、使用しないウォッチポイントを削除または無効化してから、新しいウォッチポイントをセットしてください。

print や call を使用して関数を対話的に呼び出すと、それまでにセットされていたウォッチポイントはいずれも、GDB が別の種類のブレイクポイントに到達するか、あるいは、関数の呼び出しが終了するまでの間は、効果を持たなくなります。

局所 ( 自動 ) 変数を監視するウォッチポイント、および、局所 ( 自動 ) 変数を含む式を監視するウォッチポイントは、それらの変数がスコープの範囲外となったとき、すなわち、それらの変数が定義されているブロックの外に実行制御が移ったときに、GDB が自動的に削除します。特に、デバッグ対象のプログラムが終了したときには、すべての局所変数がスコープの範囲外となるため広域変数を監視するウォッチポイントだけがセットされた状態で残ります。プログラムを再実行するのであれば、これらのウォッチポイントを再度セットする必要があります。main 関数のエントリにブレイクポイントをセットしておいて、そこに到達した後にすべてのウォッチポイントをセットするという方法があります。

注意：マルチスレッド・プログラムでは、ウォッチポイントの有用性は限定されます。現在のウォッチポイントの実装では、GDB は、単一スレッドにおいてしか式の値を監視することができません。その式の値を変更するのはカレント・スレッドだけであること ( かつ、他のスレッドがカレント・スレッドにはならないこと ) が確実であれば、通常どおりウォッチポイントを使用することができます。しかし、カレント・スレッド以外のスレッドが式の値を変更することがあると、GDB は、その変更気付かないかもしれません。

HP-UX における注意：マルチスレッド・プログラムでは、ソフトウェア・ウォッチポイントの有用性は限定されます。ソフトウェア・ウォッチポイントを生成した場合、GDB は、単一スレッドにおいてしか式の値を監視することができません。その式の値を変更するのはカレント・スレッドだけであること ( かつ、他のスレッドがカレント・スレッドにはならないこと ) が確実であれば、通常どおりソフトウェア・ウォッチポイントを使用することができます。しかし、カレント・スレッド以外のスレッドが式の値を変更

することがあると、GDB は、その変更気付かないかもしれません。(これに対して、ハードウェア・ウォッチポイントは、すべてのスレッドにおいて式を監視します。)

### 5.1.3 キャッチポイントの設定

キャッチポイントを使用することによって、C++例外や共用ライブラリのローディングのような、ある種のプログラム・イベントが発生したときに、デバグガを停止させることができます。キャッチポイントをセットするには、`catch` コマンドを使用します。

#### `catch event`

`event` で指定されるイベントが発生したときに停止します。`event` は、以下のいずれかです。

<code>throw</code>	C++例外の発生。
<code>catch</code>	C++例外のキャッチ。
<code>exec</code>	<code>exec</code> の呼び出し。現在これは、HP-UX においてのみ利用可能です。
<code>fork</code>	<code>fork</code> の呼び出し。現在これは、HP-UX においてのみ利用可能です。
<code>vfork</code>	<code>vfork</code> の呼び出し。現在これは、HP-UX においてのみ利用可能です。
<code>load</code> <code>load libname</code>	任意の共用ライブラリの動的なローディング、あるいは、 <code>libname</code> で指定されるライブラリのローディング。現在これは、HP-UX においてのみ利用可能です。
<code>unload</code> <code>unload libname</code>	動的にロードされた任意の共用ライブラリのアンローディング、あるいは、 <code>libname</code> で指定されるライブラリのアンローディング。現在これは、HP-UX においてのみ利用可能です。

#### `tcatch event`

1 回だけ停止させるキャッチポイントを設定します。最初にイベントが捕捉された後に、キャッチポイントは自動的に削除されます。

カレントなキャッチポイントの一覧を表示するには、`info break` コマンドを使用します。

現在、GDB における C++ の例外処理 (`catch throw` と `catch catch`) にはいくつかの制限があります。

- 関数に対話的に呼び出すと、GDB は通常、その関数が実行を終了したときに、ユーザに制御を戻します。しかし、その関数呼び出しが例外を発生させると、ユーザへ制御を戻すメカニズムが実行されないことがあります。この場合、ユーザ・プログラムは、アボートするか、あるいは、ブレイクポイントへの到達、GDB が監視しているシグナルの受信、そのプログラム自体の終了などのイベントが発生するまで、継続実行されることになります。これは、例外に対するキャッチポイントをセットしてある場合にもあてはまります。対話的な関数呼び出しの間は、例外に対するキャッチポイントは無効化されています。
- 対話的に例外を発生させることはできません。
- 対話的に例外ハンドラを組み込むことはできません。



`catch`コマンドが、例外処理をデバッグする手段としては最適なものではないような場合もあります。どこで例外が発生したのかを正確に知りたい場合、例外ハンドラが呼び出される前にプログラムを停止させた方がよいでしょう。なぜなら、スタック・ポインタの調整が行われる前のスタックの状態を見ることができるからです。例外ハンドラの内部にブレイクポイントを設定してしまうと、どこで例外が発生したのかを調べるのは簡単ではないでしょう。

例外ハンドラが呼び出される直前で停止させるには、実装に関する知識が若干必要になります。GNU C++の場合、以下のような ANSI C インターフェイスを持つ `__raise_exception` というライブラリ関数を呼び出すことで例外を発生させます。

```
/* addr は例外識別子が格納される領域
   id は例外識別子 */
void __raise_exception (void **addr, void *id);
```

スタック・ポインタの調整が行われる前に、すべての例外をデバッガにキャッチさせるには、`__raise_exception` にブレイクポイントを設定します (セクション 5.1 [ブレイクポイント、ウォッチポイント、キャッチポイント], ページ 31 参照)。

`id` の値に依存する条件を付けたブレイクポイント (セクション 5.1.6 [ブレイクポイントの成立条件], ページ 41 参照) を使用することで、特定の例外が発生したときにだけユーザ・プログラムを停止させることができます。複数の条件付きブレイクポイントを設定することで、複数の例外の中のどれかが発生したときにユーザ・プログラムを停止させることもできます。

#### 5.1.4 ブレイクポイントの削除

ブレイクポイント、ウォッチポイント、キャッチポイントがプログラムを停止させた後、同じところで再びプログラムを停止させたくない場合、それらを取り除くことがしばしば必要になります。これが、ブレイクポイントの削除と呼ばれるものです。削除されたブレイクポイントはもはや存在しなくなり、それが存在したという記録も残りません。

`clear` コマンドを使用する場合、ブレイクポイントを、それがプログラム内部のどこに存在するかを指定することによって削除します。`delete` コマンドの場合は、ブレイクポイント番号を指定することで、個々のブレイクポイント、ウォッチポイント、キャッチポイントを削除することができます。

ブレイクポイントで停止した後、先へ進むために、そのブレイクポイントを削除する必要はありません。ユーザが実行アドレスを変更することなく継続実行する場合、最初に実行される命令に設定されているブレイクポイントを、GDB は自動的に無視します。

`clear`        選択されているスタック・フレーム内において次に実行される命令にセットされているブレイクポイントを削除します (セクション 6.3 [フレームの選択], ページ 53 参照)。最下位にあるフレームが選択されている場合、ユーザ・プログラムが停止した箇所にセットされているブレイクポイントを削除するのに便利な方法です。

`clear function`

`clear filename: function`

`function` で指定される関数のエントリにセットされているブレイクポイントを削除します。

`clear linenum`

`clear filename: linenum`

指定された行、または、その行内に記述されたコードにセットされたブレイクポイントを削除します。

`delete [breakpoints] [range...]`

引数で指定された範囲内にあるブレイクポイント、ウォッチポイント、キャッチポイントを削除します。引数が指定されない場合、すべてのブレイクポイントを削除します ( `set confirm off` コマンドが事前に実行されていない場合、GDB は、削除してもよいかどうか確認を求めてきます )。このコマンドの省略形は `d` です。

### 5.1.5 ブレイクポイントの無効化

ブレイクポイント、ウォッチポイント、キャッチポイントを削除するのではなく、無効化したい場合もあるでしょう。無効化によってブレイクポイントは、それがあたかも削除されたかのように機能しなくなりますが、後に再度有効化することができるよう、そのブレイクポイントに関する情報は記憶されます。

ブレイクポイント、ウォッチポイント、キャッチポイントは、`enable` コマンドと `disable` コマンドによって有効化、無効化されます。これらのコマンドには、引数として 1 つ以上のブレイクポイント番号を指定することも可能です。指定すべき番号が分からない場合は、`info break` コマンド、または、`info watch` コマンドによってブレイクポイント、ウォッチポイント、キャッチポイントの一覧を表示させてください。

ブレイクポイント、ウォッチポイント、キャッチポイントは、有効 / 無効という観点から見て、4 つの異なる状態を持つことができます。

- 有効。ブレイクポイントはユーザ・プログラムを停止させます。break コマンドでセットされたブレイクポイントの初期状態はこの状態です。
- 無効。ブレイクポイントはユーザ・プログラムの実行に影響を与えません。
- 1 回有効。ブレイクポイントはユーザ・プログラムを停止させますが、停止後、そのブレイクポイントは無効状態になります。
- 1 回有効 ( 削除 )。ブレイクポイントはユーザ・プログラムを停止させますが、停止直後に、そのブレイクポイントは完全に削除されます。tbreak コマンドでセットされたブレイクポイントの初期状態はこの状態です。

以下のコマンドを使用することで、ブレイクポイント、ウォッチポイント、キャッチポイントの有効化、無効化が可能です。

`disable [breakpoints] [range...]`

指定されたブレイクポイントを無効化します。番号が 1 つも指定されない場合は、すべてのブレイクポイントが無効化されます。無効化されたブレイクポイントは何ら影響力を持ちませんが、そのブレイクポイントに関する情報まで削除されるわけではありません。そのブレイクポイントを無視する回数、ブレイクポイント成立の条件、ブレイクポイント・コマンドなどのオプションは、後にそのブレイクポイントが有効化される場合に備えて、記憶されています。disable コマンドは `dis` と省略することができます。

`enable [breakpoints] [range...]`

指定されたブレイクポイント ( または、すべての定義済みブレイクポイント ) を有効化します。有効化されたブレイクポイントは、再びユーザ・プログラムを停止させることができるようになります。

`enable [breakpoints] once range...`

指定されたブレイクポイントを一時的に有効化します。このコマンドで有効化されたブレイクポイントはどれも、最初にプログラムを停止させた直後に、GDB によって無効化されます。

`enable [breakpoints] delete range...`

1 回だけプログラムを停止させ、その直後に削除されるような設定で、指定されたブレイクポイントを有効化します。このコマンドで有効化されたブレイクポイントはどれも、最初にプログラムを停止させた直後に、GDB によって削除されます。

`tbreak` コマンド ( セクション 5.1.1 [ブレイクポイントの設定], ページ 32 参照 ) でセットされたブレイクポイントを除き、ユーザによってセットされたブレイクポイントの初期状態は有効状態です。その後、ユーザが上記のコマンドのいずれかを使用した場合に限り、無効化されたり有効化されたりします ( `until` コマンドは、独自にブレイクポイントをセット、削除することができますが、ユーザのセットした他のブレイクポイントの状態は変更しません。セクション 5.2 [継続実行とステップ実行], ページ 45 を参照してください )。

### 5.1.6 ブレイクポイントの成立条件

最も単純なブレイクポイントは、指定された箇所にプログラムが到達するたびに、プログラムの実行を停止させます。ブレイクポイントに対して条件を指定することも可能です。ここで、「条件」とは、プログラムが記述された言語で表現された真偽値を表わす式のことです ( セクション 8.1 [式], ページ 63 参照 )。条件付きのブレイクポイントにプログラムが到達するたびに、その式が評価されます。そして、その結果が真であった場合だけ、プログラムは停止します。

これは、プログラムの正当性を検査するために診断式を使用するのは逆になります。診断式の場合は、成立しないとき、すなわち条件が偽であるときに、プログラムを停止させます。C 言語で `assert` という診断式をテストするためには、しかるべきブレイクポイントに `‘! assert’` という条件を設定します。

ウォッチポイントに対して条件を設定することもできます。もともとウォッチポイントは、ある式の値を検査するものですから、これは必要ないかもしれませんが、しかし、ある変数の新しい値がある特定の値に等しいか検査するのは条件式のほうに任せて、ウォッチポイントの対象そのものは単にその変数の名前にしてしまうという設定の方が簡単でしょう。

ブレイクポイントの成立条件に副作用を持たせたり、場合によってはプログラム内部の関数を呼び出させたりすることもできます。プログラムの進行状況をログに取る関数を呼び出したり、特別なデータ構造をフォーマットして表示するユーザ定義の関数を使用したい場合などに便利です。この効果は、同じアドレスに有効なブレイクポイントが別に設定されていない限り、完全に予測可能です ( 別のブレイクポイントが設定されていると、GDB はこのブレイクポイントを先に検出し、他のブレイクポイントで設定した条件式をチェックすることなくプログラムを停止させてしまうかもしれません )。あるブレイクポイントに到達したときに副作用を持つ処理を実行させるためには、ブレイクポイントに条件を設定するよりも、ブレイクポイント・コマンドを使用する方が便利であり、柔軟でしょう ( セクション 5.1.7 [ブレイクポイント・コマンド・リスト], ページ 43 参照 )。

ブレイクポイントの成立条件は、ブレイクポイントをセットする際に、`break` コマンドの引数に `‘if’` を使用することによって設定できます。セクション 5.1.1 [ブレイクポイントの設定], ページ 32 を参照してください。ブレイクポイントの成立条件は、`condition` コマンドによっていつでも変更できます。

`watch` コマンドの中で `if` キーワードを使用することもできます。`catch` コマンドは、`if` キーワードを認識しません。キャッチポイントに対して条件を追加設定する唯一の方法は、`condition` コマンドを使うことです。

`condition bnum expression`

`bnum` で指定される番号のブレイクポイント、ウォッチポイント、キャッチポイントの成立条件として、`expression` を指定します。条件をセットした後、番号 `bnum` のブレイ

クポイントは、*expression* の値が真 (C 言語の場合はゼロ以外の値) であるときのみ、ユーザ・プログラムを停止させます。condition コマンドを使用すると、GDB はただちに *expression* の構文の正当性、および、*expression* の中で使用されるシンボル参照の、ブレイクポイントのコンテキストにおける有効性をチェックします。*expression* が、ブレイクポイントのコンテキストにおいて参照できないシンボルを使用していると、GDB は以下のようなエラー・メッセージを表示します。

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

condition コマンド (あるいは、break if ... のように条件付きでブレイクポイントをセットするコマンド) が実行されるときに、*expression* の値が GDB によって実際に評価されるわけではありません。セクション 8.1 [式], ページ 63 を参照してください。

condition *bnum*

*bnum* で指定される番号のブレイクポイントから条件を削除します。実行後、それは通常の無条件ブレイクポイントになります。

ブレイクポイント成立条件の特別なものに、ブレイクポイントに到達した回数がある数に達したときにプログラムを停止させるというものがあります。これは大変便利なので、それを実現するための特別な方法が提供されています。それは、ブレイクポイントの通過カウント (ignore count) を使用する方法です。すべてのブレイクポイントは、通過カウントと呼ばれる整数値を持っています。ほとんどの場合、この通過カウントの値はゼロであり、何ら影響力を持ちません。しかし、通過カウントとして正の値を持つブレイクポイントに到達すると、ユーザ・プログラムはそこで停止せず、単に通過カウントの値を 1 だけ減少させて処理を継続します。したがって、通過カウントが *n* であると、ユーザ・プログラムがそのブレイクポイントに到達した回数が *n* 以下の間は、そのブレイクポイントにおいてプログラムは停止しません。

ignore *bnum count*

*bnum* で指定される番号のブレイクポイントの通過カウントを *count* で指定される値にセットします。ブレイクポイントへの到達回数が *count* 以下の間、ユーザ・プログラムは停止しません。この間、GDB は、通過カウントの値を減少させる以外には何もしません。

次にブレイクポイントに到達したときにプログラムを停止させるには、*count* にゼロを指定してください。

ブレイクポイントで停止した後に continue コマンドを使用して実行を再開する場合、ignore コマンドを使用することなく、直接 continue コマンドの引数に通過カウントを指定することができます。セクション 5.2 [継続実行とステップ実行], ページ 45 を参照してください。

ブレイクポイントが通過カウントとして正の値を持ち、かつ、成立条件を持つ場合、成立条件はチェックされません。通過カウントが 0 に達すると、GDB は成立条件のチェックを再開します。

‘\$foo-- <= 0’ のように、評価のたびに値の減少するコンビニエンス変数を使用した評価式によって、通過カウントと同様の効果を達成することができます。セクション 8.9 [コンビニエンス変数], ページ 76 を参照してください。

通過カウントは、ブレイクポイント、ウォッチポイント、キャッチポイントに適用されます。

### 5.1.7 ブレイクポイント・コマンド・リスト

ブレイクポイント (あるいは、ウォッチポイント、キャッチポイント) に対して、それによってプログラムが停止したときに実行される一連のコマンドを指定することができます。例えば、ある特定の式の値を表示したり、他のブレイクポイントを有効化したりできると便利なこともあるでしょう。

```
commands [bnum]
... command-list ...
end
```

`bnum` で指定される番号を持つブレイクポイントに対して一連のコマンドを指定します。コマンド自体は、次の行以下に記述します。コマンドの記述を終了するには、`end` だけから成る 1 行を記述します。

ブレイクポイントからすべてのコマンドを削除するには、`commands` 行に続いて (コマンドを 1 つも指定せずに) `end` を記述します。

引数 `bnum` が指定されない場合、`commands` は、最後にセットされたブレイクポイント、ウォッチポイント、キャッチポイントを対象とします (最後に到達したブレイクポイントではありません)。

`command-list` の記述中は、`(RET)` キーが持つ、最後に実行されたコマンドを繰り返し実行する機能は無効です。

ブレイクポイント・コマンドを使用してプログラムの実行を再開することができます。`continue`、`step`、または、実行を再開させるその他の任意のコマンドを使用してください。

コマンド・リストの中で、実行を再開するコマンドの後に記述されているものは無視されます。というのは、プログラムが実行を再開すると (たとえそれが `next` コマンドや `step` コマンドによるものであっても) 別のブレイクポイントに到達する可能性があり、そのブレイクポイントがコマンド・リストを持っていると、どちらのリストを実行するべきかあいまいになるからです。

コマンド・リストの先頭に指定されたコマンドが `silent` であると、ブレイクポイントで停止したときに通常出力されるメッセージは表示されません。これは、ある特定のメッセージを出力して実行を継続するようなブレイクポイントをセットするのに望ましいでしょう。コマンド・リスト中の後続のコマンドがどれもメッセージを出力しない場合、ブレイクポイントに到達したことをユーザに示す情報は何も表示されないことになります。`silent` はブレイクポイント・コマンド・リストの先頭においてのみ意味を持ちます。

`echo`、`output`、`printf` の各コマンドを使用することで、細かく管理された出力を表示することができます。これらのコマンドは、`silent` 指定のブレイクポイントで使うと便利です。セクション 16.4 [制御された出力を得るためのコマンド]、ページ 165 を参照してください。

例えば、ブレイクポイント・コマンドを使用して、`foo` へのエントリにおいて `x` が正の値を持つときに、その値を表示するには以下のようにします。

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

ブレイクポイント・コマンドの 1 つの応用として、あるバグの持つ影響を取り除いて、他のバグを見つけるためにテストを継続することができます。誤りのある行の次の行に、誤りの発生を検出するような条件付きのブレイクポイントをセットし、ブレイクポイント・コマンドの中で修正の必要な変数に正しい値を割り当てます。コマンド・リストの最後には `continue` コマンドを記述して、ログ

ラムが停止しないようにします。また、プログラムの先頭には `silent` コマンドを記述し、何も出力されないようにします。以下に例を挙げます。

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

### 5.1.8 ブレイクポイント・メニュー

プログラミング言語によっては（特に C++ の場合）、異なるコンテキストにおいて使用するために、同一の関数名を複数回定義することが可能です。これは、オーバーローディングと呼ばれます。関数名がオーバーロードされている場合、`'break function'` だけでは、どこにブレイクポイントをセットしたいのかを GDB に正しく通知するのに十分ではありません。このような場合には、ブレイクポイントをセットしたい関数がどれであることを正確に指定するために、`'break function(types)'` のような形式を使用することができます。このような形式を使用しないと、GDB は候補となりえるブレイクポイントの一覧を番号付きのメニューとして表示し、プロンプト `'>'` によってユーザの選択を待ちます。先頭の 2 つの選択肢は常に、`'[0] cancel'` と `'[1] all'` です。1 を入力すると、候補となるすべての関数のそれぞれの定義に対してブレイクポイントをセットします。また、0 を入力すると、新たにブレイクポイントを設定することなく `break` コマンドを終了します。

例えば、以下に示すセッションの抜粋は、オーバーロードされたシンボル `String::after` に対してブレイクポイントをセットしようとした場合を示しています。ここでは、この関数名を持つ関数定義の中から 3 つを選択しています。

```
(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 挿入できないブレイクポイント

オペレーティング・システムによっては、他のプロセスがプログラムを実行中は、そのプログラムにおいてブレイクポイントを使用できないことがあります。このような状況においてブレイクポイントをセットしてプログラムの実行を開始したり継続したりしようとする、GDB は以下のようなエラー・メッセージを表示します。

```
Cannot insert breakpoints.
```

```
The same program may be running in another process.
```

この状況では、3つの選択肢があります。

1. ブレイクポイントを削除または無効化してから処理を継続します。
2. GDBの実行を中断し、プログラム・ファイルをコピーして新しい名前を付けます。GDBの実行を再開し、`exec-file`コマンドを使用して、新しい名前のプログラムを実行すべきことをGDBに知らせます。プログラムを再度実行します。
3. プログラムを再リンクします。その際、リンカのオプション `‘-N’` を使用して、テキスト・セグメントを共用不可とします。オペレーティング・システムの制約は、共用不可の実行ファイルの場合は無関係である可能性があります。

ハードウェアによるブレイクポイントやウォッチポイントをあまりにもたくさんアクティブにしようとする、これと似たメッセージが出力されることがあります。

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

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

このメッセージは、プログラムの実行を再開しようと試みたときに表示されます。何個のブレイクポイントとウォッチポイントが必要になるかは、GDBにもそのときまで正確には分からないからです。

このメッセージが表示された場合、ハードウェアによるブレイクポイント、ウォッチポイントをいくつか無効化または削除してから実行を継続する必要があります。

## 5.2 継続実行とステップ実行

継続実行とは、ユーザ・プログラムの実行を再開して、それが正常に終了するまで実行させることを指します。一方、ステップ実行とは、ユーザ・プログラムを1「ステップ」だけ実行することを指します。ここで「ステップ」とは、(使用されるコマンドによって)1行のソース・コードを指すこともありますし、1マシン命令を指すこともあります。継続実行の場合でもステップ実行の場合でも、ブレイクポイントやシグナルが原因となって、正常終了する前にユーザ・プログラムが停止することがあります(シグナルによってプログラムが停止した場合、実行を再開するには `handle` コマンドまたは `‘signal 0’` コマンドを使用するとよいでしょう。セクション 5.3 [シグナル], ページ 48 を参照してください)。

```
continue [ignore-count]
```

```
c [ignore-count]
```

```
fg [ignore-count]
```

ユーザ・プログラムが最後に停止した箇所から、プログラムの実行を再開します。停止箇所に設定されているブレイクポイントは無視されます。引数 `ignore-count` を指定すれば、停止箇所のブレイクポイントを無視する回数を指定することができます。これは `ignore` コマンドと似た効果を持ちます(セクション 5.1.6 [ブレイクポイントの成立条件], ページ 41 参照)。

引数 `ignore-count` は、ユーザ・プログラムがブレイクポイントによって停止した場合にのみ意味を持ちます。これ以外の場合には、`continue` コマンドへの引数は無視されます。

`c` および `fg` は、簡便さのためだけに提供されている同義コマンドで、`continue` コマンドと全く同様の動作をします(`fg` は `foreground` を省略したものです。デバッグ対象のプログラムはフォアグラウンド・プログラムであると判断されます)。

別の箇所を実行を再開するには、呼び出し関数に戻る `return` コマンド ( セクション 11.4 [関数からの復帰], ページ 107 参照 ) または、ユーザ・プログラム内の任意の箇所へ移動する `jump` コマンド ( セクション 11.2 [異なるアドレスにおける処理継続], ページ 106 参照 ) を使用することができます。

ステップ実行を使用する典型的なテクニックは、問題があると思われる関数やプログラム部分の先頭にブレイクポイント ( セクション 5.1 [ブレイクポイント、ウォッチポイント、キャッチポイント], ページ 31 参照 ) を設定し、ブレイクポイントで停止するまでプログラムを実行させた後、問題が再現するまで、関連しそうな変数の値を調べながら、疑わしい部分を 1 行ずつ実行することです。

**step**      異なるソース行に到達するまでユーザ・プログラムを継続実行した後、プログラムを停止させ、GDB に制御を戻します。このコマンドの省略形は `s` です。

注意: デバッグ情報なしでコンパイルされた関数の内部にいるときに `step` コマンドを使用すると、デバッグ情報付きの関数に達するまでプログラムの実行は継続されます。同様に、`step` コマンドがデバッグ情報なしでコンパイルされた関数の内部へ入って、停止することはありません。デバッグ情報を持たない関数の内部でステップ実行を行うには、後述の `stepi` コマンドを使用してください。

`step` コマンドは、ソース・コード行の最初の命令においてのみ停止します。これにより、`switch` 文や `for` 文などにおいて複数回停止してしまうという問題が回避されます。同じ行の中にデバッグ情報を持つ関数への呼び出しがあると、`step` コマンドは続けて停止します。すなわち、その行の中から呼び出されている関数の内部においてステップ実行を行います。

さらに `step` コマンドは、関数が行番号情報を持つ場合に限り、その関数内部に入り込みます。関数が行番号情報を持たない場合、`step` コマンドは `next` コマンドと同様の動作をします。これにより、MIPS マシン上で `cc -gl` を使用すると発生する問題が回避されます。以前のバージョンにおける `step` コマンドは、関数が何らかのデバッグ情報を持っていれば、その内部に入り込んでいました。

**step count**

`step` コマンドによるステップ実行を `count` 回繰り返します。ステップ実行を `count` 回繰り返す終わる前に、ブレイクポイントに到達するか、あるいは、ステップ実行とは関連のないシグナルが発生した場合には、ただちにステップ実行を中断して停止します。

**next [count]**

カレントな ( 最下位の ) スタック・フレーム上において、ソース・コード上の次の行まで実行します。これは `step` コマンドと似ていますが、`next` コマンドは、ソース・コード上に関数呼び出しが存在すると、その関数を停止することなく最後まで実行します。プログラムが停止するのは、`next` コマンドを実行したときと同一のスタック・フレーム上において、ソース・コード上の異なる行まで実行が継続されたときです。このコマンドの省略形は `n` です。

引数 `count` は、`step` コマンドの場合と同様、繰り返し回数です。

`next` コマンドは、ソース・コード行の最初の命令においてのみ停止します。これにより、`switch` 文や `for` 文などにおいて複数回停止してしまうという問題が回避されます。

**finish**      選択されているスタック・フレーム上の関数が復帰するまで、実行を継続します。戻り値があれば、それを表示します。

`return` コマンド ( セクション 11.4 [関数からの復帰], ページ 107 参照 ) と比較してみてください。



**until****u**

カレントなスタック・フレーム上において、カレント行よりも後ろにある行に到達するまで実行を継続します。このコマンドは、ループ内において複数回ステップ実行をするのを回避するために使用されます。これは `next` コマンドに似ていますが、唯一の相違点は、`until` コマンドによる実行によってジャンプ命令に到達した場合、プログラム・カウンタの値がジャンプ命令のアドレスより大きくなるまで、プログラムが継続実行されるという点です。

これは、ステップ実行によってループ内の最後の行に到達した後に `until` コマンドを実行することで、ループから抜け出るまでプログラムを継続実行させることができるということを意味しています。これに対して、ループ内の最後の行で `next` コマンドを実行すると、プログラムはループの先頭に戻ってしまうので、ループ内の処理を繰り返すことを余儀なくされます。

`until` コマンドの実行により、プログラムがカレントなスタック・フレームから抜け出ようとする、そこで `until` コマンドはプログラムを停止します。

実行されるマシン・コードの順序がソース行の順序と一致しない場合、`until` コマンドは直観にいくらか反するような結果をもたらすかもしれません。例えば、以下に挙げるデバッグ・セッションからの抜粋では、`f (frame)` コマンドによって、プログラムが 206 行目において停止していることが示されています。ところが、`until` コマンドを実行すると、195 行目で停止してしまいます。

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

これは、コンパイラが、実行の効率を高めるために、C 言語では `for` ループ本体の前に記述されているループ終了のための条件判定を、ループの先頭ではなく末尾で行うコードを生成したためです。この判定式にまで処理が進んだとき、`until` コマンドはあたかもループの先頭に戻ったかのように見えます。しかしながら、実際のマシン・コードのレベルでは、前の命令に戻ったわけではありません。

引数のない `until` コマンドは、1 命令ごとのステップ実行によって実現されるため、引数付きの `until` コマンドに比べて処理速度が遅くなります。

**until location****u location**

`location` で指定される箇所に到達するか、カレントなスタック・フレームを抜け出るまで、ユーザ・プログラムを継続実行します。`location` は `break` コマンドが受け付ける形式の引数です (セクション 5.1.1 [ブレイクポイントの設定], ページ 32 参照)。この形式による `until` コマンドはブレイクポイントを使用するため、引数のない `until` コマンドより処理速度が速くなります。

**stepi****stepi arg****si**

1 マシン命令を実行した後、停止してデバッガに戻ります。

マシン命令単位でステップ実行する場合、`'display/i $pc'` を使用すると便利ながしばしばあります。これは、ユーザ・プログラムが停止するたびに、次に実行される命令を GDB に自動的に表示させます。セクション 8.6 [自動表示], ページ 68 を参照してください。

引数として、`step` コマンドと同様、繰り返し回数を取ります。

```
nexti
nexti arg
ni
```

1 マシン命令を実行しますが、それが関数の呼び出しである場合は、関数から復帰するまで実行を継続します。

引数として、`next` コマンドと同様、繰り返し回数を取ります。

### 5.3 シグナル

シグナルは、プログラム内で発生する非同期イベントです。オペレーティング・システムによって、使用可能なシグナルの種類が定義され、それぞれに名前と番号が割り当てられます。例えば、UNIX においては、割り込み文字（通常は、`Ctrl` キーを押しながら `C` を押す）を入力したときにプログラムが受信する `SIGINT`、その使用領域からかけ離れたメモリ域を参照したときにプログラムが受信する `SIGSEGV`、アラームのタイムアウト時に発生する（プログラムからアラームを要求した場合にのみ発生する）`SIGALRM` シグナルなどがあります。

`SIGALRM` など、いくつかのシグナルは、プログラムの正常な機能の一部です。`SIGSEGV` などの他のシグナルは、エラーを意味します。これらのシグナルは、プログラムが事前にそれを処理する何らかの方法を指定しないと、致命的な（プログラムを即座に終了させる）ものとなります。`SIGINT` はユーザ・プログラム内部のエラーを意味するものではありませんが、通常は致命的なものであり、割り込みの目的であるプログラムの終了を実現することができます。

GDB は、ユーザ・プログラム内部における任意のシグナル発生を検出することができます。ユーザは、個々のシグナルの発生時に何を実行するかを、GDB に対して事前に指定することができます。

通常 GDB は、`SIGALRM` のようなエラーではないシグナルを無視するよう（これらのシグナルがユーザ・プログラムの中で持っている役割を妨害することのないよう）設定されています。その一方で、エラーのシグナルが発生した場合にはすぐにユーザ・プログラムを停止させるよう設定されています。これらの設定は `handle` コマンドによって変更することができます。

```
info signals
info handle
```

すべてのシグナルを一覧にして表示します。また、個々のシグナルについて、GDB がそれをどのように処理するよう設定されているかを表示します。このコマンドを使用して、定義済みのすべてのシグナルのシグナル番号を知ることができます。

`info handle` は `info signals` の別名です。

```
handle signal keywords...
```

GDB が *signal* によって指定されるシグナルを処理する方法を変更します。*signal* には、シグナル番号またはシグナル名称（先頭の '`SIG`' は省略可能）を指定します。キーワード *keywords* によって、どのように変更するかを指定します。

`handle` コマンドが受け付けるキーワードには省略形を使用することができます。省略しない場合、キーワードは以下ようになります。

```
nostop
```

GDB に対して、このシグナルが発生してもユーザ・プログラムを停止しないよう指示します。GDB は、シグナルを受信したことをメッセージ出力によってユーザに通知することができます。

```
stop
```

GDB に対して、このシグナルが発生するとユーザ・プログラムを停止するよう指示します。これは、`print` キーワードを暗黙のうちに含みます。

<code>print</code>	GDB に対して、このシグナルが発生するとメッセージを表示するよう指示します。
<code>noprint</code>	GDB に対して、このシグナルが発生したことを知らせないように指示します。これは、 <code>nostop</code> キーワードを暗黙のうちに含みます。
<code>pass</code>	GDB に対して、このシグナルの発生をユーザ・プログラムが検出できるようにするよう指示します。ユーザ・プログラムはシグナルを処理することができます。致命的なシグナルが処理されない場合、ユーザ・プログラムは停止するかもしれません。
<code>nopass</code>	GDB に対して、このシグナルの発生をユーザ・プログラムが検出できないようにするよう指示します。

シグナルによってユーザ・プログラムが停止した場合、実行を継続するまでそのシグナルは検出されません。その時点において、そのシグナルに対して `pass` キーワードが有効であれば、ユーザ・プログラムは、実行継続時にシグナルを検出します。言い換えれば、GDB がシグナルの発生を報告してきたとき、`handle` コマンドに `pass` キーワードまたは `nopass` キーワードを指定することで、実行を継続したときにプログラムにそのシグナルを検出させるか否かを制御することができます。

また、`signal` コマンドを使用することによって、ユーザ・プログラムがシグナルを検出できないようにしたり、通常は検出できないシグナルを検出できるようにしたり、あるいは任意の時点で任意のシグナルをユーザ・プログラムに検出させたりすることができます。例えば、ユーザ・プログラムが何らかのメモリ参照エラーによって停止した場合、ユーザは、さらに実行を継続しようとして、問題のある変数に正しい値を設定して継続実行しようとするかもしれません。しかし、実行継続直後に検出される致命的なシグナルのために、おそらくユーザ・プログラムはすぐに終了してしまうでしょう。このようなことを回避したければ、`'signal 0'` コマンドによって実行を継続することができます。セクション 11.3 [ユーザ・プログラムへのシグナルの通知], ページ 107 を参照してください。

## 5.4 マルチスレッド・プログラムの停止と起動

ユーザ・プログラムが複数のスレッド (セクション 4.9 [マルチスレッド・プログラムのデバッグ], ページ 26 参照) を持つ場合、すべてのスレッドにブレイクポイントを設定するか、特定のスレッドにブレイクポイントを設定するかを選択することができます。

```
break linespec thread threadno
```

```
break linespec thread threadno if ...
```

`linespec` はソース行を指定します。記述方法はいくつかありますが、どの方法を使っても結果的にはソース行を指定することになります。

`break` コマンドに修飾子 `'thread threadno'` を使用することで、ある特定のスレッドがこのブレイクポイントに到達したときだけ GDB がプログラムを停止するよう、指定することができます。ここで `threadno` は、GDB によって割り当てられるスレッド識別番号で、`'info threads'` コマンドによる出力の最初の欄に表示されるものです。

ブレイクポイントをセットする際に `'thread threadno'` を指定しなければ、そのブレイクポイントはユーザ・プログラム内部のすべてのスレッドに適用されます。

条件付きのブレイクポイントに対しても `thread` 識別子を使用することができます。この場合、以下のように `'thread threadno'` をブレイクポイント成立条件の前に記述してください。

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

いかなる理由によるのであれ GDB 配下においてユーザ・プログラムが停止した場合、カレント・スレッドだけではなく、すべての実行スレッドが停止します。これにより、知らないうちに状態の変

化が発生することを心配することなく、スレッドの切り替えも含めて、プログラム全体の状態を検査することができます。

逆に、プログラムの実行を再開したときには、すべてのスレッドが実行を開始します。これは、`step` コマンドや `next` コマンドによるシングル・ステップ実行の場合でも同様です。

特に GDB は、すべてのスレッドの歩調を合わせてシングル・ステップ実行することはできません。スレッドのスケジューリングは、デバッグ対象のマシンのオペレーティング・システムに依存する (GDB が管理するわけではない) ので、カレント・スレッドがシングル・ステップの実行を完了する前に、他のスレッドは複数の文を実行してしまうかもしれません。また、プログラムが停止するとき、他のスレッドは 2 つの文の間の境界のところまでぴったり停止するよりも、文の途中で停止してしまう方が一般的です。

また、継続実行やステップ実行の結果、プログラムが別のスレッド内で停止してしまうこともあります。最初のスレッドがユーザの要求した処理を完了する前に、他のスレッドがブレイクポイントに到達した場合、シグナルを受信した場合、例外が発生した場合には、常にこのようなことが発生します。

OS によっては、OS スケジューラをロックすることによって、ただ 1 つのスレッドだけが実行されるようにすることができます。

`set scheduler-locking mode`

スケジューラのロックング・モード (locking mode) を設定します。off の場合は、ロックのメカニズムは機能せず、任意の時点において、どのスレッドも実行される可能性を持ちます。on の場合は、再始動 (resume) されるスレッドの優先順位が低い場合には、カレント・スレッドだけが実行を継続することができます。step モードでは、シングル・ステップ実行のための最適化が行われます。ステップ実行をしている間、他のスレッドが「プロンプトを横取りする」ことがないよう、カレント・スレッドに占有権が与えられます。また、ステップ実行をしている間、他のスレッドはきわめて稀にしか (あるいは、まったく) 実行するチャンスを与えられません。next コマンドによって関数呼び出しの次の行まで処理を進めると、他のスレッドが実行される可能性は高くなります。また、`continue`、`until`、`finish` のようなコマンドを使用すると、他のスレッドは完全に自由に実行されることになります。しかし、そのタイムスライスの中でブレイクポイントに到達しない限り、他のスレッドが、デバッグの対象となっているスレッドから、GDB プロンプトを横取りすることはありません。

`show scheduler-locking`

スケジューラの現在のロックング・モードを表示します。

## 6 スタックの検査

ユーザ・プログラムが停止したとき、まず最初に、どこで停止したのか、そして、どのようにしてそこに到達したのかを知る必要があるでしょう。

ユーザ・プログラムが関数呼び出しを行うたびに、その呼び出しに関する情報が生成されます。その情報には、ユーザ・プログラム内においてその呼び出しが発生した場所、関数呼び出しの引数、呼び出された関数内部のローカル変数などが含まれます。その情報は、スタック・フレームと呼ばれるデータ・ブロックに保存されます。スタック・フレームは、呼び出しスタックと呼ばれるメモリ域に割り当てられます。

ユーザ・プログラムが停止すると、スタックを検査する GDB コマンドを使用して、この情報をすべて見ることができます。

GDB は 1 つのスタック・フレームを選択していて、多くの GDB コマンドはこの選択されたフレームを暗黙のうちに参照します。特に、GDB に対してユーザ・プログラム内部の変数の値を問い合わせると、GDB は選択されたフレームの内部においてその値を探そうとします。関心のあるフレームを選択するための特別な GDB コマンドが提供されています。セクション 6.3 [フレームの選択], ページ 53 を参照してください。

ユーザ・プログラムが停止すると、GDB はその時点において実行中のフレームを自動的に選択し、`frame` コマンド (セクション 6.4 [フレームに関する情報], ページ 54 参照) のように、そのフレームに関する情報を簡潔に表示します。

### 6.1 スタック・フレーム

呼び出しスタックは、スタック・フレーム、または短縮してフレームと呼ばれる、連続した小部分に分割されます。個々のフレームは、ある関数に対する 1 回の呼び出しに関連するデータです。フレームには、関数への引数、関数のローカル変数、関数の実行アドレスなどの情報が含まれます。

ユーザ・プログラムが起動されたとき、スタックには `main` 関数のフレームが 1 つ存在するだけです。これは、初期フレームまたは「最上位のフレーム」と呼ばれます。関数が呼び出されるたびに、新たにフレームが作成されます。関数が復帰すると、その関数を呼び出したときに生成されたフレームが取り除かれます。関数が再帰的に呼び出される場合、1 つの関数に対して多くのフレームが生成されるということもありえます。実際に実行中の関数に対応するフレームは、「最下位のフレーム」と呼ばれます。これは、存在するすべてのスタック・フレームの中で、最も新しく作成されたものです。

ユーザ・プログラムの内部においては、スタック・フレームはアドレスによって識別されます。スタック・フレームは多くのバイトから構成され、それぞれがそれ自身のアドレスを持っています。どのような種類のコンピュータにおいても、これらのバイトのうちの 1 つのバイトのアドレスをもってフレームのアドレスとする慣習的な方法が提供されています。通常、あるフレーム内部で実行中は、そのフレームのアドレスがフレーム・ポインタ・レジスタと呼ばれるレジスタに格納されています。

GDB は、既存のスタック・フレームのすべてに番号を割り当てます。最下位のフレームは 0 で、それを呼び出したフレームは 1 となります。以下、最下位のフレームを起点として、順番に値を割り当てていきます。これらの番号はユーザ・プログラム内部には実際には存在しません。これらの番号は、GDB コマンドでスタック・フレームを指定することができるように、GDB によって割り当てられたものです。

コンパイラによっては、スタック・フレームを使用せずに実行されるように関数をコンパイルする方法を提供しているものもあります (例えば、`gcc` のオプション

`'-fomit-frame-pointer'`)

を指定すると、フレームを持たない関数が生成されます)。これは、フレームをセットアップする時間を節約するために、頻繁に利用されるライブラリ関数に対してしばしば適用されます。これらの関数の呼び出しを処理するために GDB が提供する機能は限られています。最下位のフレームの関数呼び出しがスタック・フレームを持たない場合、GDB は、あたかもそれが通常どおりに番号 0 のフレームを持つものとみなして、関数呼び出しの連鎖を追跡できるようにします。しかしながら、最下位以外のスタック位置に存在する、フレームを持たない関数に対しては、GDB は特別な処置を取りません。

`frame args`

`frame` コマンドによって、あるスタック・フレームから別のスタック・フレームに移動し、選択したスタック・フレームを表示させることができます。`args` は、フレームのアドレスまたはスタック・フレーム番号です。引数なしで実行すると、`frame` コマンドはカレントなスタック・フレームを表示します。

`select-frame`

`select-frame` コマンドによって、フレームを表示することなく、あるスタック・フレームから別のスタック・フレームに移動することができます。これは、`frame` コマンドから、表示処理を取り除いたものです。

## 6.2 バックトレース

バックトレースとは、ユーザ・プログラムが現在いる箇所にどのようにして到達したかを示す要約情報です。複数のフレームが存在する場合、1 フレームの情報を 1 行に表示します。現在実行中のフレーム (番号 0 のフレーム) を先頭に、それを呼び出したフレーム (番号 1 のフレーム) を次行に、以降、同様にスタックをさかのぼって情報を表示します。

`backtrace`

`bt` 全スタックのバックトレースを表示します。スタック内のすべてのフレームが、1 行に 1 フレームずつ表示されます。

システムの割り込み文字 (通常は、`Ctrl` キーを押しながら `C` を押す) によって、いつでもバックトレースを停止することができます。

`backtrace n`

`bt n` 引数のない `backtrace` コマンドと似ていますが、最下位のフレームから数えて  $n$  個のフレームだけが表示されます。

`backtrace -n`

`bt -n` 引数のない `backtrace` コマンドと似ていますが、最上位のフレームから数えて  $n$  個のフレームだけが表示されます。

`backtrace` の別名としては、ほかに `where` や `info stack` (省略形は `info s`) があります。

`backtrace` コマンドの出力結果の各行に、フレーム番号と関数名が表示されます。`set print address off` コマンドを実行していなければ、プログラム・カウンタの値も表示されます。`backtrace` コマンドの出力結果では、関数への引数に加えて、ソース・ファイル名や行番号も表示されます。プログラム・カウンタが、行番号で指定される行の最初のコードを指している場合、その値は省略されます。

以下に `backtrace` の例を示します。これは、`'bt 3'` の出力であり、したがって最下位のフレームから 3 フレームが表示されています。

```
#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...)
```

番号 0 のフレームを表示する行の先頭には、プログラム・カウンタの値がありません。これは、builtin.cの 993行目の最初のコードにおいてユーザ・プログラムが停止したことを表わしています。

### 6.3 フレームの選択

スタックやユーザ・プログラム内の他のデータを調べるためのほとんどのコマンドは、それが実行された時点において選択されているスタック・フレーム上で動作します。以下に、スタック・フレームを選択するためのコマンドを列挙します。どのコマンドも、それによって選択されたスタック・フレームに関する簡単な説明を最後に表示します。

frame *n*

*f n*      番号 *n* のフレームを選択します。最下位の ( 現在実行中の ) フレームが番号 0 のフレーム、最下位のフレームを呼び出したフレームが番号 1 のフレーム、以下同様となります。最も大きい番号を持つフレームは main のフレームです。

frame *addr*

*f addr*      アドレス *addr* のフレームを選択します。スタック・フレームの連鎖がバグのために破壊されてしまって、GDB がすべてのフレームに正しく番号を割り当てられないような場合に、この方法が役に立ちます。さらに、ユーザ・プログラムが複数のスタックを持ち、スタックの切り替えを行うような場合にも有効です。

SPARC アーキテクチャでは、フレームを任意に選択するには、フレーム・ポインタ、スタック・ポインタの 2 つのアドレスを frame に指定する必要があります。

MIPS、Alpha の両アーキテクチャでは、スタック・ポインタ、プログラム・カウンタの 2 つのアドレスが必要です。

29k アーキテクチャでは、レジスタ・スタック・ポインタ、プログラム・カウンタ、メモリ・スタック・ポインタの 3 つのアドレスが必要です。

*up n*      スタックを *n* フレームだけ上へ移動します。*n* が正の値の場合、最上位のフレームに向かって移動します。これは、より大きいフレーム番号を持ち、より長く存在しているフレームへの移動です。*n* のデフォルト値は 1 です。

*down n*      スタックを *n* フレームだけ下へ移動します。*n* が正の値の場合、最下位のフレームに向かって移動します。これは、より小さいフレーム番号を持ち、より最近作成されたフレームへの移動です。*n* のデフォルト値は 1 です。down の省略形は do です。

これらのコマンドはいずれも、最後にフレームに関する情報を 2 行で表示します。1 行めには、フレーム番号、関数名、引数、ソース・ファイル名、そのフレーム内において実行停止中の行番号が表示されます。2 行めには、実行停止中のソース行が表示されます。

以下に、例を示します。

```
(gdb) up
#1  0x22f0 in main (argc=1, argv=0xf7ffbf4, env=0xf7ffbf4)
    at env.c:10
10      read_input_file (argv[i]);
```

この情報が表示された後で、`list` コマンドを引数なしで実行すると、フレーム内で実行停止中の行を中心に 10 行のソース行が表示されます。セクション 7.1 [ソース行の表示], ページ 57 を参照してください。

`up-silently n`

`down-silently n`

これら 2 つのコマンドは、それぞれ、`up` コマンド、`down` コマンドの変種です。相違点は、ここに挙げた 2 つのコマンドが、新しいフレームに関する情報を表示することなく実行されるという点にあります。これらは、情報の出力が不必要で邪魔ですらある、GDB のコマンド・スクリプトの中での使用を主に想定したものです。

## 6.4 フレームに関する情報

既に挙げたもの以外にも、選択されたスタック・フレームに関する情報を表示するコマンドがいくつかあります。

`frame`

`f` このコマンドは、引数なしで実行されると、別のフレームを選択するのではなく、その時点において選択中のフレームに関する簡単な説明を表示します。このコマンドの省略形は `f` です。引数付きの場合、このコマンドはスタック・フレームを選択するのに使用されます。セクション 6.3 [フレームの選択], ページ 53 を参照してください。

`info frame`

`info f` このコマンドは、選択されたスタック・フレームに関する詳細な情報を表示します。表示される情報には、以下のようなものがあります。

- フレームのアドレス
- 1 つ下位の ( 選択されたフレームによって呼び出された ) フレームのアドレス
- 1 つ上位の ( 選択されたフレームを呼び出した ) フレームのアドレス
- 選択されたフレームに対応するソース・コードを記述した言語
- フレームの引数のアドレス
- フレームのローカル変数のアドレス
- フレームに退避されているプログラム・カウンタ ( 呼び出し側フレームの実行アドレス )
- フレームに退避されているレジスタ

これらの詳細な情報は、何か問題が発生して、スタックの形式が通常の慣習に合致しなくなった場合に、役に立ちます。

`info frame addr`

`info f addr`

アドレス `addr` のフレームに関する詳細な情報を、そのフレームを選択することなく表示します。このコマンドによって、その時点において選択されていたフレームとは異なるフレームが選択されてしまうことはありません。このコマンドでは、`frame` コマンドに指定するのと同様のアドレスを ( アーキテクチャによっては複数 ) 指定する必要があります。セクション 6.3 [フレームの選択], ページ 53 を参照してください。

`info args` 選択中のフレームの引数を、1 行に 1 つずつ表示します。



`info locals`

選択中のフレームのローカル変数を、1 行に 1 つずつ表示します。これらはすべて、選択中のフレームの実行箇所においてアクセス可能な（静的変数または自動変数として宣言された）変数です。

`info catch`

選択中のスタック・フレームの実行箇所においてアクティブな状態にある、すべての例外ハンドラの一覧を表示します。他の例外ハンドラを参照したい場合は、関連するフレームに（`up` コマンド、`down` コマンド、`frame` コマンドを使用して）移動してから、`info catch` を実行します。セクション 5.1.3 [キャッチポイントの設定], ページ 38 を参照してください。



## 7 ソース・ファイルの検査

GDB は、ユーザ・プログラムのソース・コードの一部を表示することができます。これは、プログラムの中に記録されているデバッグ情報によって、そのプログラムをビルドするのに使用されたソース・ファイルを GDB が知るができるからです。ユーザ・プログラムが停止すると、GDB は自動的にプログラムが停止した行を表示します。同様に、ユーザがあるスタック・フレーム（セクション 6.3 [フレームの選択], ページ 53 参照）を選択すると、そのフレームにおいて実行が停止している行を GDB は表示します。明示的にコマンドを使用することで、ソース・ファイルの他の部分を表示することも可能です。

GNU Emacs インターフェイス経由で GDB を使用しているユーザは、Emacs の提供する機能を使ってソース・ファイルを参照する方を好むかもしれません。これについては、章 17 [Using GDB under GNU Emacs], ページ 167 を参照してください。

### 7.1 ソース行の表示

ソース・ファイル内の行を表示するには、`list` コマンド（省略形は `l`）を使用します。デフォルトでは、10 行が表示されます。ソース・ファイルのどの部分を表示するかを指定する方法がいくつかあります。

最もよく使われる `list` コマンドの形式を以下に示します。

`list linenum`

現在のソース・ファイルの行番号 *linenum* を中心に、その前後の行を表示します。

`list function`

関数 *function* の先頭を中心に、その前後の行を表示します。

`list`        ソース・ファイル行の続きを表示します。既に表示された最後の行が `list` コマンドによって表示されたのであれば、その最後の行の次の行以降が表示されます。しかし、既に表示された最後の行が、スタック・フレーム（章 6 [スタックの検査], ページ 51 参照）の表示の一部として 1 行だけ表示されたのであれば、その行の前後の行が表示されます。

`list -`      前回表示された行の前に位置する行を表示します。

`list` コマンドを上記の形式のいずれかによって実行すると、GDB はデフォルトでは 10 行のソース行を表示します。これは `set listsize` コマンドによって変更することができます。

`set listsize count`

`list` コマンドで表示される行数を *count* に設定します（`list` コマンドの引数で他の値が明示的に指定された場合は、この設定は効力を持ちません）。

`show listsize`

`list` コマンドが表示する行数を表示します。

`list` コマンドを実行後、`(RET)` キーによって `list` コマンドを実行した場合、引数は破棄されます。したがって、これは単に `list` と入力して実行したのと同じことになります。同じ行が繰り返し表示されるよりも、この方が役に立つでしょう。ただし、引数 `-` は例外となります。この引数は繰り返し実行の際にも維持されるので、繰り返し実行することで、ソース・ファイルの内容がさかのぼって表示されていきます。

一般的には、`list` コマンドは、ユーザによって 0 個、1 個、または 2 個の行指定（*linespec*）が与えられることを期待しています。ここで行指定とは、ソース行を指定するものです。いくつかの記

述方法がありますが、いずれも結果的には何らかのソース行を指定するものです。list コマンドの引数として使用できるものの完全な説明を以下に示します。

`list linespec`  
*linespec* によって指定される行を中心に、その前後の行を表示します。

`list first,last`  
*first* 行から *last* 行までを表示します。両引数はいずれも行指定です。

`list ,last` *last* 行までを表示します。

`list first,`  
*first* 行以降を表示します。

`list +` 最後に表示された行の次の行以降を表示します。

`list -` 最後に表示された行の前の行以前を表示します。

`list` 前述のとおり。

以下に、ソースの特定の 1 行を指定する方法を示します。これは、いずれも行指定です。

*number* 現在のソース・ファイルの行番号 *number* の行を指定します。list コマンドの引数に 2 つの行指定がある場合、2 つめの行指定は、最初の行指定と同一のソース・ファイルを指定します。

`+offset` 最後に表示された行から *offset* で指定される行数だけ下にある行を指定します。2 つの行指定を引数として持つ list コマンドにおいて、これが 2 つめの行指定として使用される場合、最初の行指定から *offset* で指定される行数だけ下の行を指定します。

`-offset` 最後に表示された行から *offset* で指定される行数だけ上にある行を指定します。

`filename: number`  
 ソース・ファイル *filename* の行番号 *number* の行を指定します。

`function` 関数 *function* の本体の先頭行を指定します。例えば C 言語では、左括弧 (‘{’) のある行を指します。

`filename: function`  
 ファイル *filename* 内の関数 *function* の本体を開始する左括弧 (‘{’) のある行を指定します。異なるソース・ファイルの中に同一の名前の関数が複数ある場合にのみ、あいまいさを回避するために、関数名とともにファイル名を指定する必要があります。

`*address` プログラム・アドレス *address* を含む行を指定します。*address* には任意の式を指定することができます。

## 7.2 ソース・ファイル内の検索

カレントなソース・ファイル内において正規表現による検索を行うためのコマンドが 2 つあります。

`forward-search regexp`

`search regexp`

‘forward-search *regexp*’ コマンドは、最後に list コマンドによって表示された行の 1 つ下の行から、1 行ずつ正規表現 *regexp* による検索を行います。正規表現にマッチするものが見つかったら、その行を表示します。‘search *regexp*’ という同義のコマンドを使うこともできます。コマンド名は、省略して `fo` とすることができます。

`reverse-search regexp`

‘`reverse-search regexp`’コマンドは、最後に `list` コマンドによって表示された行の 1 つ上の行から、1 行ずつ逆方向に向かって正規表現 `regexp` による検索を行います。正規表現にマッチするものが見つかったら、その行を表示します。コマンド名は、省略して `rev` とすることができます。

### 7.3 ソース・ディレクトリの指定

実行形式プログラムは、それがコンパイルされたソース・ファイルの名前だけを記録して、ソース・ファイルの存在するディレクトリ名を記録しないことがあります。また、ディレクトリ名が記録された場合でも、コンパイル時とデバッグ時との間に、そのディレクトリが移動してしまっている可能性があります。GDB は、ソース・ファイルを検索すべきディレクトリの一覧を持っています。これは、ソース・パスと呼ばれます。GDB は、ソース・ファイルが必要なときにはいつでも、それが見つかるまで、このリストの中のすべてのディレクトリを、リストの中に記述されている順に探します。実行ファイルのサーチ・パスは、この目的では使用されないことに気をつけてください。またカレントな作業ディレクトリも、それがたまたまソース・パスの中にある場合を除けば、この目的で使用されることはありません。

GDB がソース・パスの中でソース・ファイルを見つけることができない場合、プログラムがディレクトリ名を記録してあれば、そのディレクトリも検索されます。ソース・パスにディレクトリの指定がなく、コンパイルされたディレクトリの名前も記録されていない場合、GDB は最後の手段としてカレント・ディレクトリを探します。

ソース・パスを空にした場合、または、再調整した場合、ソース・ファイルを見つけた場所や個々の行のファイル内の位置のような、GDB が内部でキャッシュしている情報は消去されます。

GDB を起動した時点では、ソース・パスには ‘`cdir`’ と ‘`cwd`’ だけが指定されています。‘`cdir`’ のほうが前に指定されています。他のディレクトリをソース・パスに追加するには、`directory` コマンドを使用してください。

`directory dirname ...`

`dir dirname ...`

ディレクトリ `dirname` をソース・パスの先頭に追加します。個々のディレクトリをコロン ‘`:`’ (MS-DOS と MS-Windows では ‘`;`’)。これらの環境では通常、‘`:`’ は絶対ファイル名の中で使用されます) または、空白で区切ることによって、複数のディレクトリをこのコマンドに渡すことができます。ソース・パスの中に既に存在するディレクトリを指定することもできます。この場合、そのディレクトリの、ソース・パスの中での位置が前に移動するので、GDB はそのディレクトリを以前よりも早く検索することになります。

(コンパイル時のディレクトリが記録されていれば) それを指すのに文字列 ‘`$cdir`’ を使うことができます。また、カレントな作業ディレクトリを指すには、文字列 ‘`$cwd`’ を使うことができます。‘`$cwd`’ と ‘`.`’ (ピリオド) とは同じではありません。前者は、GDB セッション内においてカレントな作業ディレクトリが変更された場合、変更されたディレクトリを指します。これに対して後者は、ソース・パスへの追加を行ったときに、その時点におけるカレント・ディレクトリに展開されてしまいます。

`directory`

ソース・パスの内容を再び空にします。ソース・パスを空にする前に、確認を求めてきます。

`show directories`

ソース・パスを表示します。ソース・パスに含まれるディレクトリ名を見ることができます。

ソース・パスの中に、不要となってしまったディレクトリが混在していると、GDB が誤ったバージョンのソースを見つけてしまい、混乱をもたらすことがあります。以下の手順によって、正常な状態にすることができます。

1. ソース・パスを空にするために、`directory` コマンドを引数なしで実行します。
2. ソース・パス中に含めたいディレクトリが組み込まれるよう、`directory` コマンドに適切な引数を指定して実行します。すべてのディレクトリを、1 回のコマンド実行で追加することができます。

## 7.4 ソースとマシン・コード

`info line` コマンドを使用してソース行をプログラム・アドレスに（あるいは、プログラム・アドレスをソース行に）対応付けすることができます。また、`disassemble` コマンドを使用して、あるアドレス範囲をマシン命令として表示することもできます。GNU Emacs モードで実行されている場合、`info line` コマンドは指定された行を示す矢印を表示します。また、`info line` コマンドは、アドレスを 16 進形式だけではなくシンボリック形式でも表示します。

`info line linespec`

ソース行 *linespec* に対応するコンパイル済みコードの開始アドレス、終了アドレスを表示します。`list` コマンド（セクション 7.1 [ソース行の表示], ページ 57 参照）が理解できる任意の形式によってソース行を指定することができます。

例えば、`info line` コマンドによって、関数 `m4_changequote` の最初の行に対応するオブジェクト・コードの位置を知ることができます。

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

また、（*linespec* の形式として *\*addr* を使用することで）ある特定のアドレスがどのソース行に含まれているのかを問い合わせることができます。

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

`info line` の実行後、`x` コマンドのデフォルト・アドレスは、その行の先頭アドレスに変更されます。これにより、マシン・コードの調査を開始するには '`x/i`' を実行するだけで十分となります（セクション 8.5 [メモリの調査], ページ 67 参照）。また、このアドレスはコンビニエンス変数 `$_` の値として保存されます（セクション 8.9 [コンビニエンス変数], ページ 76 参照）。

`disassemble`

この特殊コマンドは、あるメモリ範囲をマシン命令としてダンプ出力します。デフォルトのメモリ範囲は、選択されたフレームにおいてプログラム・カウンタが指している箇所を含む関数です。このコマンドに引数を 1 つ渡すと、それはプログラム・カウンタ値を指定することになります。GDB は、その値が指す箇所を含んでいる関数をダンプ出力します。2 つの引数を渡すと、ダンプ出力するアドレス範囲（1 つめのアドレスは含まれますが、2 つめのアドレスは含まれません）を指定することになります。

以下の例は、あるアドレス範囲の HP PA-RISC 2.0 コードを逆アセンブルした結果を示しています。

```
(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.
```

アーキテクチャによっては、一般に使用される命令ニーモニックを複数持つものや、異なる構文を持つものがあります。

`set disassembly-flavor instruction-set`

`disassemble` コマンドまたは `x/i` コマンドによってプログラムの逆アセンブルを行う際に使用する命令セットを選択します。

現在のところ、このコマンドは、Intel x86 ファミリに対してのみ定義されています。*instruction-set* は、`intel` と `att` のいずれかにセットすることができます。デフォルトは `att` です。これは、x86 ベースのターゲット用の UNIX アセンブラがデフォルトで使用している AT&T 仕様のスタイルです。





## 8 データの検査

ユーザ・プログラムの中のデータを調べる通常の方法は、`print` コマンド ( 省略形は `p` )、またはそれと同義のコマンドである `inspect` コマンドを使用することです。これは、ユーザ・プログラムが記述された言語 ( 章 9 [異なる言語の使い方], ページ 79 参照 ) による式を評価し、その値を出力するものです。

```
print expr
print /f expr
```

`expr` は ( ソース言語による ) 式です。デフォルトでは、`expr` の値は、`expr` のデータ型にとって適切な形式で表示されます。‘/f’を指定することで、他の形式を選択することも可能です。‘/f’の `f` は形式を指定する文字です。セクション 8.4 [出力フォーマット], ページ 66 を参照してください。

```
print
print /f
```

`expr` を省略すると、GDB は値履歴 ( セクション 8.8 [値履歴], ページ 75 参照 ) の最後の値を再表示します。これは、同じ値を異なる形式で調べるのに便利です。

データを調べるためのより低レベルの方法は、`x` コマンドを使うことです。これは、指定されたアドレスのメモリ上のデータを、指定された形式で表示するものです。セクション 8.5 [メモリの調査], ページ 67 を参照してください。

型に関する情報に関心があるとき、また、構造体やクラスのフィールドがどのように宣言されているかという点に関心があるときは、`print` コマンドではなく `ptype expr` コマンドを使用してください。章 10 [シンボル・テーブルの検査], ページ 101 を参照してください。

### 8.1 式

`print` コマンド、および、ほかの多くの GDB コマンドは、式を受け取って、その値を評価します。ユーザの使用しているプログラミング言語によって定義されている定数、変数、演算子は、いずれも GDB における式の中で有効です。これには、条件式、関数呼び出し、キャスト、文字列定数が含まれます。しかし、プリプロセッサの `#define` コマンドによって定義されるシンボルは、残念ながら含まれません。

GDB は、ユーザの入力する式において配列定数をサポートします。その構文は {*element*, *element*...} です。例えば、コマンド `print {1, 2, 3}` を使用して、ターゲット・プログラム内で `malloc()` によって獲得されたメモリ内に配列を作成することができます。

C 言語は大変広汎に使用されているので、このマニュアルの中で示される例の中のほとんどの式は C 言語で記述されています。他の言語での式の使い方に関する情報については、章 9 [異なる言語の使い方], ページ 79 を参照してください。

この節では、プログラミング言語によらず GDB の式で利用できる演算子を説明します。

キャストは、C 言語のみならず、すべての言語でサポートされています。これは、メモリ内のあるアドレスにある構造体を調べるのに、数値をポインタにキャストするのが大変便利であるからです。

プログラミング言語によらず共通に使用可能な演算子に加えて、GDB は以下の演算子をサポートしています。

- @        ‘@’ は、メモリの一部を配列として処理するための 2 項演算子です。詳細については、セクション 8.3 [人工配列], ページ 65 を参照してください。
- ::        ‘::’ によって、それを定義している関数またはファイルを特定して、変数を指定することができます。セクション 8.2 [プログラム変数], ページ 64 を参照してください。

`{type} addr`

`addr` で示されるメモリ上のアドレスに格納されている、`type` で示される型のオブジェクトを参照します。`addr` には、評価結果が整数値またはポインタになるような任意の式を指定することができます（ただし、2 項演算子の前後には、キャストを使う場合と同様の括弧が必要です）。これは、`addr` の位置に通常存在するデータの型がいかなるものであろうとも、使用することができます。

## 8.2 プログラム変数

最も一般的に使用される式は、ユーザ・プログラム内部の変数名です。

式の中の変数は、選択されたスタック・フレーム（セクション 6.3 [フレームの選択], ページ 53 参照）内において解釈されます。これは、以下の 2 つのいずれかとなります。

- グローバル変数（または、ファイル・スコープの静的変数）

あるいは

- プログラム言語のスコープ規則によって、そのフレームの実行中の箇所から可視の変数

つまり、以下の例において、ユーザ・プログラムが関数 `foo` を実行中は、変数 `a` を調べたり使用したりすることができますが、変数 `b` を使用したり調べたりすることができるのは、`b` が宣言されているブロックの内部をユーザ・プログラムが実行中である場合に限られます。

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

ただし、これには 1 つ例外があります。特定の 1 ソース・ファイルをスコープとする変数や関数は、たとえ現在の実行箇所がそのファイルの中ではなくても、参照することができます。しかし、このような変数または関数が（異なるソース・ファイル中に）同じ名前で複数個存在するということがありえます。このような場合、その名前を参照すると予期できない結果をもたらします。2 つのコロンを並べる記法によって、特定の関数またはファイルの中の静的変数を指定することができます。

```
file::variable
function::variable
```

ここで `file` または `function` は、静的変数 `variable` のコンテキスト名です。ファイル名の場合は、引用符を使用することによって、GDB がファイル名を確実に 1 つの単語として解釈するようにさせることができます。例えば、ファイル `'f2.c'` の中で定義されたグローバル変数 `x` の値を表示するには、

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

このような `::` の用途が、これと非常によく似ている C++ における `::` の用途と衝突することは非常に稀です。GDB は、式の内部において C++ のスコープ解決演算子の使用もサポートしています。

注意：ときどき、新しいスコープに入った直後やスコープから出る直前に、関数内部の特定の箇所から見ると、ローカル変数の値が正しくないように見ることがあります。

マシン命令単位でステップ実行を行っているときに、このような問題を経験することがあるかもしれません。これは、ほとんどのマシンでは、（ローカル変数定義を含む）スタック・フレームのセット

アップに複数の命令が必要となるからです。マシン命令単位でステップ実行を行う場合、スタック・フレームが完全に構築されるまでの間は、変数の値が正しくないように見えることがあります。スコープから出るときには、スタック・フレームを破棄するのに、通常複数のマシン命令が必要とされます。それらの命令群の中をステップ実行し始めた後には、ローカル変数の定義は既に存在しなくなっているかもしれません。

このようなことは、コンパイラが重要な最適化を実施する場合にも、発生する可能性があります。常に正確な値が見えることを確実にするためには、コンパイルの際に、すべての最適化を行わないようにします。

コンパイラの最適化の結果として起こりえる別の可能性として、使用されない変数が存在しなくなることや、変数が (メモリ・アドレスではなく) レジスタに割り当てられることがあります。コンパイラの使用するデバッグ情報形式がこのような状況に対してどのようなサポートを提供しているかに応じて、場合によっては、そのようなローカル変数の値を GDB が表示できないということがありえます。この場合、GDB は次のようなメッセージを表示します。

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

このような問題を解決するには、最適化をせずに再コンパイルするか、あるいは、コンパイラがいくつかのデバッグ情報形式をサポートしているのであれば、異なるデバッグ情報形式を使用します。例えば、GNU C/C++ コンパイラである GCC は通常、`-gstabs` オプションをサポートしています。`-gstabs` オプションは、COFF などよりも優れた形式でデバッグ情報を生成します。また、別の効率的なデバッグ情報形式である DWARF-2 (`-gdwarf-2`) を使用することもできます。詳細については、セクション “Options for Debugging Your Program or GNU CC” in *Using GNU CC* を参照してください。

### 8.3 人工配列

メモリ内に連続的に配置されている同一型のオブジェクトを表示することが役に立つことがよくあります。配列の一部や動的にサイズの決定される配列にアクセスするのに、そこへのポインタしかプログラム内部に存在しないような場合です。

これは、2 項演算子 `@` を使用して、連続したメモリ範囲を人工配列として参照することで可能です。`@` の左側のオペランドは、参照したい配列の最初の要素で、かつ、1 個のオブジェクトでなければなりません。また、右側のオペランドは、その配列の中の参照したい部分の長さでなければなりません。結果は、その要素がすべて左側の引数と同型である配列の値です。第 1 の要素は左側の引数そのものです。第 2 の要素は、第 1 の要素を保持するメモリ域の直後のメモリ上から取られます。これ以降の要素も同様です。以下に例を示します。プログラムが以下のようになっているとしましょう。

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

以下を実行することで、array の内容を表示することができます。

```
p *array@len
```

`@` の左側のオペランドは、メモリ上に実在するものでなければなりません。このような方法で `@` によって作成された配列の値は、配列の添字付けの見地からは他の配列と同様に振る舞い、式の中で使用された場合は強制的にポインタとして扱われます。人工配列は、一度表示された後、値履歴 (セクション 8.8 [値履歴], ページ 75 参照) を通して式の中に現れることがよくあります。

人工配列を作成するもう 1 つの方法は、キャストを使用することです。これによって、ある値を配列として解釈し直します。この値は、メモリ上に実在するものでなくてもかまいません。

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

ユーザの便宜を考慮して、(例えば、`(type[])value`のように)配列の長さが省略された場合その値を満たすサイズを(`sizeof(value)/sizeof(type)`のように)GDBが計算します。

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

ときには、人工配列の機構では十分でないことがあります。かなり複雑なデータ構造では、関心のある要素が連続的に並んでいないことがあります。例えば、配列の中のポインタの値に関心がある場合です。このような状況において役に立つ回避策の1つに、関心のある値のうち最初のもを表示する式の中のカウンタとしてコンビニエンス変数(セクション 8.9 [コンビニエンス変数], ページ 76 参照)を使用し、`(RET)`キーによってその式を繰り返し実行することです。例えば、構造体へのポインタの配列 `dtab`があり、個々の構造体のフィールド `fv`の値に関心があるとしましょう。以下に、この場合の例を示します。

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

## 8.4 出力フォーマット

デフォルトでは、GDBはデータの型にしたがって値を表示します。ときには、これが望ましくない場合もあります。例えば、数値を16進で表示したい場合やポインタを10進で表示したい場合があるでしょう。あるいは、メモリ内のある特定のアドレスのデータを文字列や命令として表示させたい場合もあるでしょう。このようなことをするためには、値を表示するとき出力フォーマットを指定します。

出力フォーマットの最も単純な使用法は、既に評価済みの値の表示方法を指定することです。これは、`print`コマンドの最初の引数をスラッシュとフォーマット文字で開始することで行います。サポートされているフォーマット文字は、以下のとおりです。

- x            値を整数値とみなし、16進で表示します。
  - d            値を符号付き10進の整数値として表示します。
  - u            値を符号なし10進の整数値として表示します。
  - o            値を8進の整数値として表示します。
  - t            値を2進の整数値として表示します。‘t’はtwoを省略したものです。<sup>1</sup>
  - a            値を、16進の絶対アドレス、および、そのアドレスより前にあるシンボルのうち最も近い位置にあるものからのオフセット・アドレスとして表示します。このフォーマットを使用することで、未知のアドレスがどこに(どの関数の中に)あるのかを知ることができます。
- ```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```
- c            値を整数値とみなし、文字定数として表示します。

<sup>1</sup> 原注: ‘b’はフォーマット文字として使用できません。フォーマット文字はxコマンドでも共通して使用されますが、xコマンドでは、‘b’はbyteの省略形として使用されているためです。セクション 8.5 [メモリの調査], ページ 67 を参照してください。

**f** 値を浮動小数点数値とみなし、典型的な浮動小数点の構文で出力します。

例えば、プログラム・カウンタの値を 16 進数で表示する ( セクション 8.10 [レジスタ], ページ 77 参照 ) には、以下を実行してください。

```
p/x $pc
```

スラッシュの前にはスペースが必要ではないことに注意してください。これは、GDB のコマンド名にはスラッシュを含めることができないからです。

値履歴の最後の値を異なる形式で再表示するには、`print` コマンドに対して式を指定せずにフォーマットだけを指定して実行します。例えば、`'p/x'` を実行すると最後の値を 16 進で再表示します。

## 8.5 メモリの調査

コマンド `x` ( `examine` の `x` ) を使用することで、ユーザ・プログラム内のデータ型にかかわらず、メモリ上の値をいくつかの形式で調べることができます。

```
x/nfu addr
```

```
x addr
```

**x** メモリ上の値を調べるには `x` コマンドを使用してください。

`n`、`f`、`u` はいずれも、どれだけのメモリをどのようにフォーマットして表示するかを指定するための、必須ではないパラメータです。`addr` は、メモリの表示を開始するアドレスを指定する式です。`nfu` の部分にデフォルトを使用するのであれば、スラッシュ '/' は必要ありません。いくつかのコマンドによって、`addr` に対して便利なデフォルト値を指定することができます。

**n** ( 繰り返し回数 )

繰り返し回数は 10 進の整数値です。デフォルトは 1 です。これによって、( 単位 `u` の ) メモリをどれだけ表示するかを指定します。

**f** ( 表示フォーマット )

表示フォーマットには、`print` コマンドによって使用されるフォーマット、`'s'` ( NULL 文字で終了する文字列 )、`'i'` ( マシン命令 ) のいずれかを指定します。初期状態では、デフォルトは `'x'` ( 16 進 ) です。デフォルトは、`x` コマンドまたは `print` コマンドを実行するたびに変更されます。

**u** ( メモリ・サイズの単位 )

単位の大きさは以下のいずれかになります。

**b** バイト

**h** ハーフ・ワード ( 2 バイト )

**w** ワード ( 4 バイト ) —これが初期状態のデフォルトです。

**g** ジャイアント・ワード ( 8 バイト )

`x` コマンド実行時に単位の大きさを指定するたびに、その大きさが、次に `x` コマンドを実行する際のデフォルトになります ( フォーマット `'s'` および `'i'` については、単位の大きさは無視されます。これらについては通常、単位の大きさを指定しません )。

**addr** ( 表示を開始するアドレス )

`addr` は、GDB にメモリの表示を開始させたいアドレスです。この式は、必ずしもポインタ値を持つ必要はありません ( ポインタ値を持つことも可能です )。これは常に、メモ

リ内のある1バイトを指す整数値のアドレスとして解釈されます。式に関する詳細については、セクション 8.1 [式], ページ 63 を参照してください。addr のデフォルトは通常、最後に調べられたアドレスの次のアドレスになります。しかし、ほかのコマンドによってもデフォルトのアドレスが設定されます。該当するコマンドは、info breakpoints (デフォルトは、最後に表示されたブレイクポイントのアドレスに設定されます)、info line (デフォルトは、行の先頭アドレスに設定されます)、および、print コマンド (メモリ内の値を表示するのに使用した場合) です。

例えば、`'x/3uh 0x54320'` は、先頭アドレス 0x54320 から始めて、メモリ上の3個のハーフ・ワード (h) の値を、符号なし10進整数値 ('u') としてフォーマットして表示するよう求める要求です。また、`'x/4xw $sp'` は、スタック・ポインタ ('\$sp') については、セクション 8.10 [レジスタ], ページ 77 参照) の上位4ワード ('w') のメモリの内容を16進 ('x') で表示します。

単位の大きさを示す文字と出力フォーマットを指定する文字とは異なるので、単位の大きさとフォーマットのどちらが前にくるべきかを記憶しておく必要はありません。どちらを先に記述しても動作します。`'4xw'` という出力指定と `'4wx'` という出力指定とは、全く同一の意味を持ちます (ただし、繰り返し回数 *n* は最初に指定しなければなりません。`'wx4'` ではうまく動きません)。

単位の大きさ *u* は、フォーマット '*s*' および '*i*' については無視されますが、繰り返し回数 *n* を使用したいことがあるかもしれません。例えば、`'3i'` はオペランドも含めて3つのマシン命令を表示したいということを指定しています。disassemble コマンドは、マシン命令を調べる別の方法を提供してくれます。セクション 7.4 [Source and machine code], ページ 60 を参照してください。

x コマンドへの引数のデフォルトはすべて、x コマンドを使用してメモリ上を連続的に参照するために最少の情報だけを指定すればよいように設計されています。例えば、`'x/3i addr'` によって3個のマシン命令を調べた後、`'x/7'` とするだけで、続く7個のマシン命令を調べることができます。`(RET)` キーによって x コマンドを繰り返し実行する場合は、前回の繰り返し回数 *n* が再度使用されます。その他の引数も、後続の x コマンド使用時のデフォルトになります。

x コマンドによって表示されるアドレスや内容は、値履歴に保存されません。これらの数がしばしば膨大になり、邪魔になるからです。その代わりに GDB は、これらの値をコンビニエンス変数 `$_` および `$_` の値として、後続の式の内部で使用するようにします。x コマンドを実行後、最後に調べられたアドレスは、コンビニエンス変数 `$_` の値として式の中で使用することができます。また、GDB によって調べられたそのアドレスの内容は、コンビニエンス変数 `$_` の値として使用可能です。

x コマンドに繰り返し回数が指定されている場合、保存されるのは、最後に表示されたメモリ単位のアドレスとその内容です。これは、最後の出力行にいくつかのメモリ単位が表示されている場合は、最後に表示されたアドレス値と一致しません。

## 8.6 自動表示

ある1つの式の値を (それがどのように変化するかを見るために) 頻繁に表示したい場合は、その式を自動表示リストに加えて、ユーザ・プログラムが停止するたびに、GDB がその値を表示するようにするとよいでしょう。リストに加えられた個々の式には、それを識別するための番号が割り当てられます。ある式をリストから削除する際に、その番号を指定します。自動表示は、例えば以下のように表示されます。

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

ここでは、項目番号、式、および、その式の現在の値が表示されます。x コマンドや print コマンドによって手作業で表示を要求する場合と同様、好みの出力フォーマットを指定することができます。実は、display コマンドは、ユーザのフォーマットの指定の詳細度によって、print コマンドと x コマ

ンドのいずれを使用するかを決定しています。単位の大きさが指定された場合や、xコマンドでしかサポートされていない2つのフォーマット(‘i’と‘s’)のいずれかが指定された場合には、xコマンドが使用されます。それ以外の場合は、printコマンドが使用されます。

#### display *expr*

ユーザ・プログラムが停止するたびに表示される式のリストに、式 *expr* を追加します。セクション 8.1 [式], ページ 63 を参照してください。

コマンドの実行後に(RET)キーを押しても、displayコマンドは繰り返し実行されません。

#### display/*fmt* *expr*

*fmt* の部分に、大きさや繰り返し回数は指定せず、出力フォーマットだけを指定した場合は、式 *expr* を自動表示リストに追加して、出力時のフォーマットが常に、指定されたフォーマット *fmt* になるよう調整します。セクション 8.4 [出力フォーマット], ページ 66 を参照してください。

#### display/*fmt* *addr*

*fmt* の部分に‘i’、‘s’を指定した場合、あるいは、単位の大きさ、単位の数を指定した場合は、ユーザ・プログラムが停止するたびに調べるメモリ・アドレスとして式 *addr* を追加します。ここで「調べる」というのは、実際には‘x/*fmt* *addr*’を実行することを意味します。セクション 8.5 [メモリの調査], ページ 67 を参照してください。

例えば、‘display/i \$pc’は、ユーザ・プログラムが停止するたびに、次に実行されるマシン命令を見るのに便利です(‘\$pc’は、プログラム・カウンタを指すのに一般に使用される名前です。セクション 8.10 [レジスタ], ページ 77 参照)。

#### undisplay *dnums*...

##### delete display *dnums*...

表示すべき式のリストから、項目番号 *dnums* に対応する要素を削除します。

undisplayコマンドを実行後に(RET)キーを押しても、コマンドは再実行されません(仮に再実行されてしまうとすると、‘No display number ...’というエラーになるだけです)。

#### disable display *dnums*...

項目番号 *dnums* の表示を不可にします。表示不可にされた表示項目は自動的に表示されませんが、削除されたわけではありません。後に、表示可能にすることができます。

#### enable display *dnums*...

項目番号 *dnums* の表示を可能にします。これにより、表示不可が指定されるまで、式の自動表示が再度有効になります。

display リスト上の式のカレントな値を表示します。これは、ユーザ・プログラムが停止したときに実行されるのと同じ処理です。

#### info display

自動的に表示されるよう設定された式のリストを表示します。個々の式の項目番号は表示されますが、値は表示されません。このリストには、表示不可になっている式も含まれ、そのことが分かるようにマーク付けされています。また、表示されるリストには、その時点ではアクセスできない自動変数を参照しているために、その時点では値を表示することのできない式も含まれます。

表示される式がローカル変数への参照を含む場合、そのローカル変数がセットアップされているコンテキストの範囲外では、その式は無意味です。このような式は、その中の変数の1つでも定義されないコンテキストが実行開始されると表示不可になります。例えば、引数 `last_char` を取る関数の内部で `display last_char` コマンドを実行すると、その関数の内部でユーザ・プログラムが実行を停止し続ける間は、GDB はこの引数を表示します。ほかの箇所 (`last_char` という変数が存在しない箇所) で停止したときには、自動的に表示不可となります。次にユーザ・プログラムが `last_char` が意味を持つ箇所では、再びその式の表示を可能にすることができます。

## 8.7 表示設定

GDB は、配列、構造体、シンボルをどのように表示するかを制御するための方法を提供しています。

これらの設定は、どのプログラミング言語で記述されたプログラムのデバッグにも便利です。

`set print address`

`set print address on`

これにより GDB は、スタック・トレース、構造体の値、ポインタの値、ブレイクポイントなどの内容を表示する場合でも、それらの位置を示すアドレスをあわせて表示します。デフォルトは `on` です。例として、`set print address on` のときのスタック・フレームの表示結果を示します。

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

`set print address off`

アドレスの内容を表示するときには、そのアドレスを表示しません。例えば、`set print address off` のときに前の例と同一のスタック・フレームを表示すると、以下のようになります。

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

‘`set print address off`’を使用することで、GDB のインターフェイスからマシンに依存する表示をすべて取り除くことができます。例えば、`print address off` を指定してあれば、ポインタ引数の有無にかかわらず、すべてのマシン上において同一のバックトレース情報を得るはずで。

`show print address`

アドレスが表示されるか否かを示します。

GDB がシンボリックなアドレスを表示する際には通常、そのアドレスの前にある最も近い位置のシンボルと、そのシンボルからのオフセットを表示します。そのシンボルによってアドレスが一意に決まらない場合 (例えば、単一のソース・ファイルを有効範囲とする名前である場合) には、確認の必要があるかもしれません。1つの方法は、例えば ‘`info line *0x4537`’ のように、`info line` コマンドを実行することです。または、シンボリックなアドレスを表示するときに、一緒にソース・ファイルや行番号を表示するよう GDB を設定する方法もあります。



```
set print symbol-filename on
```

シンボリックな形式のアドレスの表示において、シンボルのソース・ファイル名と行番号を表示するよう、GDB に通知します。

```
set print symbol-filename off
```

シンボルのソース・ファイル名と行番号を表示しません。これがデフォルトです。

```
show print symbol-filename
```

シンボリックな形式でのアドレス表示において、GDB がそのシンボルのソース・ファイル名と行番号を表示するか否かを示します。

シンボルのソース・ファイル名と行番号を表示するのが役に立つもう 1 つの状況として、コードを逆アセンブルする場合があります。GDB が、個々の命令に対応する行番号とソース・ファイルを表示してくれます。

また、アドレスをシンボリック形式で表示させるのは、そのアドレスと、そのアドレスより前にあるシンボルのうちそのアドレスに最も近い位置にあるものが、距離的に適度に接近している場合に限定させたいこともあるかもしれません。

```
set print max-symbolic-offset max-offset
```

アドレスと、そのアドレスより前にある最も近いシンボルの間のオフセットが *max-offset* 未満のときのみ、そのアドレスをシンボリックな形式で表示するよう GDB に通知します。デフォルトは 0 で、これは GDB に対して、アドレスより前にシンボルがある場合には、常にそのアドレスをシンボリックな形式で表示するよう通知します。

```
show print max-symbolic-offset
```

GDB がシンボリックなアドレスを表示する上限となる、最大のオフセット値を問い合わせます。

あるポインタがどこを指しているか定かではない場合には、`'set print symbol-filename on'`を試みてください。こうすれば、`'p/a pointer'`を使用して、そのポインタが指している変数の名前とソース・ファイル上の位置が分かります。これは、アドレスをシンボリック形式で解釈します。例えば以下の例では、ある変数 `ptt` がファイル `'hi2.c'` 内で定義された別の変数 `t` を指していることを、GDB が教えてくれています。

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

注意：ローカル変数を指すポインタについては、たとえ適切な `set print` オプションが有効になっていても、`'p/a'` はそのポインタによって参照される変数のシンボル名やファイル名を表示しません。

異なる種類のオブジェクトについては、他の設定によって表示方法が制御されます。

```
set print array
```

```
set print array on
```

配列をきれいに表示します。このフォーマットは読むのには便利ですが、より多くのスペースを取ります。デフォルトは `off` です。

```
set print array off
```

配列を詰め込み形式で表示します。

`show print array`

配列の表示方法として、詰め込み形式ときれいな形式のどちらが選択されているかを示します。

`set print elements number-of-elements`

GDB によって表示される配列の要素の数に上限を設定します。GDB が大きな配列を表示している際に、表示された要素の数が `set print elements` コマンドで設定された数に達すると、そこで表示が停止されます。この上限は、文字列の表示にも適用されます。GDB の起動時に、この上限は 200 にセットされます。*number-of-elements* に 0 をセットすると、要素は無制限に表示されます。

`show print elements`

大きな配列を表示する際に GDB が表示する要素数を示します。0 の場合、表示される要素数に制限はありません。

`set print null-stop`

最初に NULL が検出された時点で、GDB に文字配列の表示を停止させます。これは、大きな配列が実際には短い文字列しか含んでいないときに役に立ちます。デフォルトは off です。

`set print pretty on`

構造体を表示する際に、インデントされた形式で 1 行に 1 メンバずつ GDB に表示させます。以下に例を示します。

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

`set print pretty off`

構造体を詰め込み形式で GDB に表示させます。以下に例を示します。

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

これがデフォルトの形式です。

`show print pretty`

GDB が、構造体を表示するのにどちらの形式を使用しているかを示します。

`set print sevenbit-strings on`

7 ビット文字だけを使用して表示します。このオプションがセットされていると、GDB は ( 文字列内または単一文字内の ) 8 ビット文字を `\nnn` という表記法で表示します。この設定は、英語 ( ASCII ) 環境において、文字の最上位ビットをマーカーや「メタ」ビットとして使用する場合に最適です。

`set print sevenbit-strings off`

8 ビット文字を表示します。これにより文字セットの使用が国際的になります。これがデフォルトです。

`show print sevenbit-strings`

GDB が 7 ビット文字だけを表示するか否かを示します。

`set print union on`

GDB に対して、構造体の中に含まれている共用体を表示するよう通知します。これがデフォルトの設定です。

`set print union off`

GDB に対して、構造体の中に含まれている共用体を表示しないよう通知します。

`show print union`

GDB に対して、構造体の中に含まれている共用体を表示するか否かを問い合わせます。

例えば、以下のように宣言されている場合、

```
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}};
```

`set print union on`が有効な場合、`'p foo'`は以下のような表示を行います。

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

また、`set print union off`が有効な場合、`'p foo'`は以下のような表示を行います。

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

以下の設定は、C++プログラムをデバッグしているときに関係があります。

`set print demangle`

`set print demangle on`

C++のシンボル名を、型セーフ ( type-safe ) なリンクのためにアセンブラ、リンカに渡されるエンコードされた ( mangled ) 形式ではなく、ソースに記述された形式で表示します。デフォルトは on です。

`show print demangle`

C++のシンボル名が、エンコードされた ( mangled ) 形式、ソース ( demangled ) 形式のいずれの形式で表示されるかを示します。

`set print asm-demangle`

`set print asm-demangle on`

C++のシンボル名を、命令の逆アセンブル時のようにアセンブラ・コードで表示しているときにも、エンコードされた ( mangled ) 形式ではなく、ソース形式で表示します。デフォルトは off です。

`show print asm-demangle`

アセンブラ・コードの表示において、C++シンボル名をエンコードされた ( mangled ) 形式、ソース ( demangled ) 形式のいずれの形式で表示するかを示します。

`set demangle-style style`

C++シンボル名を表現するためにさまざまなコンパイラによって使用されるいくつかのエンコーディング方式の中から 1 つを選択します。現在 *style* として選択可能であるものを以下に列挙します。

|                    |                                                                                                                                                                        |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>auto</code>  | GDB がユーザ・プログラムを解析してデコーディング方式を決定することを許します。                                                                                                                              |
| <code>gnu</code>   | GNU C++ ( <code>g++</code> ) エンコーディング・アルゴリズムに基づいてデコードします。これがデフォルトです。                                                                                                   |
| <code>hp</code>    | HP ANSI C++ ( <code>aCC</code> ) エンコーディング・アルゴリズムに基づいてデコードします。                                                                                                          |
| <code>lucid</code> | Lucid C++ ( <code>lcc</code> ) エンコーディング・アルゴリズムに基づいてデコードします。                                                                                                            |
| <code>arm</code>   | <i>C++ Annotated Reference Manual</i> に記述されているアルゴリズムを使用してデコードします。注意：この設定だけでは、 <code>cfront</code> によって生成された実行モジュールをデバッグするのに十分ではありません。これを可能にするためには、GDB をさらに拡張する必要があります。 |

*style* を指定しないと、指定可能なフォーマットの一覧が表示されます。

`show demangle-style`

C++シンボルをデコードするのに現在使用されているエンコーディング方式を示します。

`set print object`

`set print object on`

オブジェクトへのポインタを表示する際に、仮想関数テーブルを使用して、宣言された型ではなく、オブジェクトの実際の ( 派生された ) 型を表示します。

`set print object off`

仮想関数テーブルは参照せず、オブジェクトの宣言された型だけを表示します。これがデフォルトの設定です。

`show print object`

オブジェクトの実際の型と宣言された型のどちらが表示されるかを示します。

`set print static-members`

`set print static-members on`

C++のオブジェクトを表示する際、静的メンバを表示します。デフォルトは `on` です。

`set print static-members off`

C++のオブジェクトを表示する際、静的メンバを表示しません。

`show print static-members`

C++の静的メンバが表示されるか否かを示します。

`set print vtbl`

`set print vtbl on`

C++の仮想関数テーブルをきれいな形式で表示します。デフォルトは `off` です。( `vtbl` コマンドは、HP ANSI C++コンパイラ ( `aCC` ) によってコンパイルされたプログラムに対しては、機能しません。)

```
set print vtbl off
```

C++の仮想関数テーブルをきれいな形式で表示しません。

```
show print vtbl
```

C++の仮想関数テーブルをきれいな形式で表示するか否かを示します。

## 8.8 値履歴

printコマンドにより表示された値は、GDBの値履歴に保存されます。これによりユーザは、これらの値をほかの式の中で参照することができます。値は、シンボル・テーブルが（例えば、fileコマンドや symbol-fileコマンドにより）再読み込みされるか破棄されるまで維持されます。シンボル・テーブルが変更されると、値履歴が破棄されるのは、その中の値が、シンボル・テーブル内で定義されている型を参照しているかもしれないからです。

表示される値は履歴番号を与えられ、この番号によって参照することができます。この番号は1から始まる連続した整数です。printコマンドは、値に割り当てられた履歴番号を、値の前に '\$num = 'という形で表示します。ここで、numがその履歴番号です。

値履歴の中の任意の値を参照するには、'\$'に続けて履歴番号を指定します。printコマンドが出力に付加するラベルは、ユーザにこのことを知らせるためのものです。\$単体では、履歴内の最も新しい値を参照し、\$\$はその1つ前の値を参照します。\$\$nは、最新のものから数えてn番目の値を参照します。\$\$2は\$\$の1つ前の値を参照し、\$\$1は\$\$と同一、\$\$0は\$と同一です。

例えば、ユーザがたった今、構造体へのポインタを表示し、今度はその構造体の内容を見たいと考えているとしましょう。この場合は、

```
p *$
```

を実行すれば十分です。また、連結された構造体があり、そのメンバの nextが次の構造体を指すポインタであるとする、次の構造体の内容を表示するには、

```
p *$.next
```

とします。このように連結された構造体を次々に表示するには、このコマンドを繰り返し実行すればよく、それは(RET)キーによって可能です。

この履歴は、式ではなく、値を記録するという点に注意してください。xの値が4のときに、以下のコマンドを実行すると、printコマンドによって値履歴に記録される値は、xの値が変化したにもかかわらず4のままです。

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

```
show values
```

値履歴内の最新の10個の値を、項目番号付きで表示します。これは、'p \$\$9'を10回実行するようなものですが、両者の違いは、show valuesが履歴を変更しないという点にあります。

```
show values n
```

値履歴内の項目番号nを中心に、その前後の10個の値を表示します。

```
show values +
```

値履歴内の値のうち最後に表示されたものの直後にある10個の値を表示します。値が存在しない場合には、何も表示されません。

show values nを繰り返し実行するのに(RET)キーを押すことは、'show values +'を実行するのと全く同じ結果をもたらします。

## 8.9 コンビニエンス変数

GDB のコンビニエンス変数は、GDB 中である値を保持しておいて、それを後に参照するという目的で使うことができます。これらの変数は、GDB 内部においてのみ存在するものです。それらはユーザ・プログラムの中に存在するものではなく、コンビニエンス変数を設定してもユーザ・プログラムの実行には直接影響を与えません。したがって、ユーザはこれを自由に使うことができます。

コンビニエンス変数名は、先頭が '\$' で始まります。'\$' で始まる名前は、あらかじめ定義されたマシン固有のレジスタ名 (セクション 8.10 [レジスタ], ページ 77 参照) と一致しない限り、コンビニエンス変数の名前として使うことができます (これに対して、値履歴の参照名では '\$' に続けて番号を記述します。セクション 8.8 [値履歴], ページ 75 を参照してください)。

ユーザ・プログラムの中で変数に値を設定するのと同じように、代入式を使用してコンビニエンス変数に値を保存することができます。例えば、`object_ptr` が指すオブジェクトが保持する値を `$foo` に保存するには、以下のようにします。

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

コンビニエンス変数は、最初に使用されたときに生成されますが、新しい値を割り当てるまで、その値は空 (void) です。値は、いつでも代入することによって変更可能です。

コンビニエンス変数には決まった型はありません。コンビニエンス変数には、既に異なる型のデータが割り当てられている場合でも、構造体や配列を含めた任意の型のデータを割り当てることができます。コンビニエンス変数は、式として使用される場合には、その時点における値の型を持ちます。

`show convenience`

それまでに使用されたコンビニエンス変数とその値の一覧を表示します。省略形は、`show conv` です。

コンビニエンス変数の 1 つの使い方に、インクリメントされるカウンタや先へ進んでいくポインタとしての使い方があります。例えば、構造体配列の中の連続する要素のあるフィールドの値を表示したい場合、以下のコマンドを `(RET)` キーで繰り返し実行します。

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

GDB によって、いくつかのコンビニエンス変数が自動的に作成され、役に立ちそうな値が設定されます。

`$_`      `$_` 変数には、`x` コマンドによって最後に調べられたアドレスが自動的に設定されます (セクション 8.5 [メモリの調査], ページ 67 参照)。`x` コマンドによって調べられるデフォルトのアドレスを提供する他のコマンドも、`$_` にそのアドレスを設定します。このようなコマンドには、`info line` や `info breakpoint` があります。`$_` の型は、`x` コマンドによって設定された場合は `$_` の型へのポインタであり、それ以外の場合は `void *` です。

`$_`      `$_` 変数には、`x` コマンドによって最後に調べられたアドレス位置にある値が自動的に設定されます。型は、データが表示されたフォーマットに適合するように選択されます。

`$_exitcode`

`$_exitcode` 変数には、デバッグされているプログラムが終了した際の終了コードが自動的に設定されます。

HP-UX システムでは、ドル記号で始まる関数名や変数名を指定すると、GDB は、コンビニエンス変数を探す前に、まずユーザ名やシステム名を探します。

## 8.10 レジスタ

マシン・レジスタの内容は、先頭が '\$' で始まる名前を持つ変数として、式の中で参照することができます。レジスタの名前は、マシンによって異なります。info registers コマンドを使用することで、そのマシンで使用されているレジスタの名前を知ることができます。

```
info registers
```

( 選択されたスタック・フレームにおける ) 浮動小数点レジスタを除くすべてのレジスタの名前と値を表示します。

```
info all-registers
```

浮動小数点レジスタも含めてすべてのレジスタの名前と値を表示します。

```
info registers regname ...
```

指定されたレジスタ *regname* の相対化された値 ( *relativized value* ) を表示します。以下に詳しく述べるように、レジスタの値は、通常は、選択されたスタック・フレームと関係を持つ相対的な値です。*regname* には、ユーザの使用しているマシン上において有効な任意のレジスタ名が設定可能です。先頭の '\$' は、あってもなくてもかまいません。

GDB は、そのマシン・アーキテクチャが持つレジスタの正規のニーモニックと衝突しない限り、ほとんどのマシン上 ( の式の中 ) において利用可能な、4 つの「標準的」なレジスタ名を持っています。レジスタ名 \$pc と \$sp は、プログラム・カウンタ・レジスタとスタック・ポインタを指すために使われます。\$fp は、カレントなスタック・フレームへのポインタを保持するレジスタを指すために使われます。\$ps は、プロセッサの状態を保持するレジスタを指すために使われます。例えば、プログラム・カウンタの値を 16 進数で表示するには、以下のように実行します。

```
p/x $pc
```

また、次に実行される命令を表示するには、以下のように実行します。

```
x/i $pc
```

さらに、スタック・ポインタに 4 を加える<sup>2</sup> には、以下のように実行します。

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

可能な場合にはいつでも、これら 4 つの標準的なレジスタ名が使用可能です。ユーザのマシンが異なる正規のニーモニックを使用している場合でも、名前の衝突さえ起こらなければ、使用可能です。info registers コマンドにより、正規名を見ることができます。例えば、SPARC 上で info registers コマンドを実行すると、プロセッサ・ステータス・レジスタは \$psr と表示されますが、このレジスタを \$ps として参照することもできます。また、x86 ベースのマシン上では、\$ps は EFLAGS レジスタの別名となっています。

レジスタがこの方法で調べられるとき、GDB は普通のレジスタの内容を常に整数値とみなします。マシンによっては、浮動小数点値以外を保持できないレジスタを持つものがあります。このようなレジスタは、浮動小数点値を持つものとみなされます。普通のレジスタの内容を浮動小数点値として参照する方法はありません ( 'print/f \$regname' により、浮動小数点値として値を表示することはできます )。

<sup>2</sup> 原注: これは、スタックがメモリの下位方向に伸長するマシン ( 最近のほとんどのマシンがそうです ) 上において、スタックから 1 ワードを取り除く方法です。これは、最下位のスタック・フレームが選択されていることを想定しています。これ以外のスタック・フレームが選択されているときには、\$sp に値を設定することは許されません。マシン・アーキテクチャに依存することなくスタックからフレーム全体を取り除くには、return を使用します。セクション 11.4 [関数からの復帰], ページ 107 を参照してください。

レジスタには、raw と virtual の 2 つの異なるデータ形式を取るものがあります。これは、オペレーティング・システムによってレジスタの内容が保存されときのデータ形式が、ユーザ・プログラムが通常認識しているものと同じではないことを意味しています。例えば、68881 浮動小数点コプロセッサのレジスタの値は常に extended ( raw ) 形式で保存されていますが、C 言語によるプログラムは通常 double ( virtual ) 形式を想定しています。このような場合、GDB は通常 ( ユーザ・プログラムにとって意味のある形式である ) virtual 形式だけを扱いますが、info registers コマンドはデータを両方の形式で表示してくれます。

通常、レジスタの値は、選択されたスタック・フレーム ( セクション 6.3 [フレームの選択], ページ 53 参照 ) と関係を持つ相対的な値です。これは、ユーザにレジスタの値として見えるものは、選択されたフレームから呼び出されているすべてのスタック・フレームが終了し、退避されたレジスタの値が復元されたときに、そのレジスタが持つであろう値です。ハードウェア・レジスタの本当の値を知りたいければ、最下位のフレームを ( 'frame 0' で ) 選択しなければなりません。

しかし、GDB は、コンパイラが生成したコードから、どこにレジスタが保存されているかを推論する必要があります。退避されていないレジスタがある場合や、GDB が退避されたレジスタを見つけることができない場合は、どのスタック・フレームを選択していても結果は同じです。

### 8.11 浮動小数ハードウェア

構成によっては、GDB は浮動小数ハードウェアの状態について、より詳しい情報を提供することができます。

info float

浮動小数ユニットに関するハードウェア依存の情報を表示します。浮動小数チップの種類によって、表示内容やレイアウトは変わります。現在、'info float' は ARM マシンと x86 マシンにおいてサポートされています。



## 9 異なる言語の使い方

異なるプログラミング言語であっても共通点があるのが普通ですが、その表記法が全く同様であるということはめったにありません。例えば、ポインタ `p` の指す値を取り出す方法は、ANSI C では `*p` ですが、Modula-2 では `p^` です。値の表現方法（および表示方法）もまた異なります。16 進数は、C では `'0x1ae'` のようになりますが、Modula-2 では `'1AEH'` のようになります。

いくつかの言語については、言語固有の情報が GDB に組み込まれており、これにより、プログラムを記述した言語を使って上記のような操作を記述したり、プログラムを記述した言語の構文にしたがって GDB に値を出力させることができます。式を記述するのに使用される言語を、作業言語と呼びます。

### 9.1 ソース言語の切り替え

作業言語を制御する方法は 2 つあります。GDB に自動的に設定させる方法と、ユーザが手作業で選択する方法です。どちらの目的でも、`set language` コマンドを使用することができます。起動時のデフォルトでは、GDB が言語を自動的に設定するようになっています。作業言語は、ユーザの入力する式がどのように解釈されるか、あるいは、値がどのように表示されるかを決定します。

この作業言語とは別に、GDB の認識しているすべてのソース・ファイルには、それ自体の作業言語があります。オブジェクト・ファイルのフォーマットによっては、ソース・ファイルの記述言語を示す情報を、コンパイラが書き込んでいることがあるかもしれません。しかし、ほとんどの場合、GDB はファイル名から言語を推定します。ソース・ファイルの言語の種類が、C++ シンボル名がデコード (demangle) されるか否かを制御します。これにより `backtrace` は、個々のフレームを、その対応する言語にしたがって適切に表示することができます。GDB の中から、ソース・ファイルの言語を設定することはできません。しかし、ファイル名の拡張子と言語の間の関連付けを設定することはできます。セクション 9.2 [言語の表示], ページ 81 を参照してください。

他の言語で記述されたソースから C のソースを生成する、`cfront` や `f2c` のようなプログラムをユーザが使用する場合には、このことが問題となるでしょう。このような場合には、生成される C の出力に `#line` 指示子を使用するよう、そのプログラムを設定してください。こうすることによって、GDB は、元になったプログラムのソース・コードが記述された言語を正しく知ることができ、生成された C のコードではなく、元になったソース・コードを表示します。

#### 9.1.1 ファイル拡張子と言語のリスト

ソース・ファイル名が以下のいずれかの拡張子を持つ場合、GDB はその言語を以下に示すものと推定します。

|                       |                  |
|-----------------------|------------------|
| <code>' .c '</code>   | C ソース・ファイル       |
| <code>' .C '</code>   |                  |
| <code>' .cc '</code>  |                  |
| <code>' .cp '</code>  |                  |
| <code>' .cpp '</code> |                  |
| <code>' .cxx '</code> |                  |
| <code>' .c++ '</code> | C++ ソース・ファイル     |
| <code>' .f '</code>   |                  |
| <code>' .F '</code>   | Fortran ソース・ファイル |

```

‘.ch’
‘.c186’
‘.c286’    CHILL ソース・ファイル

‘.mod’    Modula-2 ソース・ファイル

‘.s’
‘.S’      アセンブラ言語のソース・ファイル。この場合、実際の動作はほとんど C 言語と同様ですが、ステップ実行時に、関数呼び出しのための事前処理部を GDB はスキップしません。

```

さらに、言語に対してファイル名の拡張子を関連付けすることも可能です。セクション 9.2 [言語の表示], ページ 81 を参照してください。

### 9.1.2 作業言語の設定

GDB に言語を自動的に設定させる場合、ユーザのデバッグ・セッションとユーザのプログラムにおいて、式は同様に解釈されます。

もしそうしなければ、言語を手作業で設定することもできます。そのためには、コマンド `‘set language lang’` を実行します。ここで、*lang* は、*c* や *modula-2* のような言語名です。サポートされている言語のリストは、`‘set language’` で表示させることができます。

言語を手作業で設定すると、GDB は、作業言語を自動的に更新することができなくなります。このことは、作業言語がソースの言語と同一ではなく、かつ、ある式がどちらの言語でも有効でありながらその意味が異なるような状況でプログラムをデバッグしようとしたときに、混乱をもたらす可能性があります。例えば、カレントなソース・ファイルが C 言語で記述されていて、GDB がそれを Modula-2 として解析している場合に、

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

のようなコマンドを実行すると、その結果は意図したものとは異なるものになるでしょう。これは C 言語では、*b* と *c* とを加算して、その結果を *a* に入れるということを意味し、表示される結果は、*a* の値となります。Modula-2 では、これは *a* と *b+c* の結果を比較して BOOLEAN 型の値を出力することを意味します。

### 9.1.3 GDB によるソース言語の推定

GDB に作業言語を自動的に設定させるには、`‘set language local’` または `‘set language auto’` を使用します。この場合、GDB は作業言語を推定します。つまり、ユーザ・プログラムが（通常はブレークポイントに達することによって）あるフレーム内部で停止したとき、GDB は、そのフレーム内の関数に対して記録されている言語を作業言語として設定します。フレームの言語が不明の場合（つまり、そのフレームに対応する関数またはブロックが、既知ではない拡張子を持つソース・ファイルにおいて定義されている場合）カレントな作業言語は変更されず、GDB は警告メッセージを出力します。

このようなことは、全体がただ 1 つの言語で記述されているほとんどのプログラムにおいては不要であると思われるでしょう。しかし、あるソース言語で記述されたプログラム・モジュールやライブラリは、他のソース言語で記述されたメイン・プログラムから使用することができます。このような場合に `‘set language auto’` を使用することで、作業言語を手作業で設定する必要がなくなります。

## 9.2 言語の表示

以下のコマンドは、作業言語、および、ソース・ファイルの記述言語を知りたいときに役に立ちます。

`show language`

カレントな作業言語を表示します。printコマンドなどによってユーザ・プログラム内部の変数を含む式を作成したり評価したりするには、このコマンドによって示される言語を使用します。

`info frame`

選択されているフレームのソース言語を表示します。このフレームの中の識別子を使用すると、この言語が作業言語になります。このコマンドにより表示される他の情報について知りたい場合は、セクション 6.4 [フレームに関する情報], ページ 54 を参照してください。

`info source`

選択されているソース・ファイルのソース言語を表示します。このコマンドにより表示される他の情報のことを知りたい場合は、章 10 [シンボル・テーブルの検査], ページ 101 を参照してください。

普通ではない状況においては、標準のリストに含まれない拡張子を持つソース・ファイルがあるかもしれません。この場合には、その拡張子を特定の言語に明示的に関連付けすることができます。

`set extension-language .ext language`

拡張子 `.ext` を持つソース・ファイルは、ソース言語 `language` によって記述されているものと想定するよう設定します。

`info extensions`

すべてのファイル拡張子と、その拡張子に関連付けされた言語を一覧表示します。

## 9.3 型と範囲のチェック

注意: 現在のリリースでは、型チェックと範囲チェックを行う GDB コマンドは組み込まれていますが、それらは実際には何も実行しません。このセクションでは、これらのコマンドが本来持つべく意図されている機能について記述してあります。

いくつかの言語は、一連のコンパイル時チェック、実行時チェックによって、一般によく見られるエラーの発生を防ぐように設計されています。これらのチェックには、関数や演算子への引数の型のチェックや、数学的操作の結果のオーバーフローを実行時に確実に検出することなどが含まれています。このようなチェックは、型の不一致を排除したり、ユーザ・プログラムの実行時に範囲エラーをチェックしたりすることによって、コンパイル後のプログラムの正しさを確かなものにするのに役に立ちます。

GDB は、ユーザが望むのであれば、上記のような条件のチェックを行います。GDB はユーザ・プログラムの文をチェックすることはありませんが、例えば、printコマンドによる評価を目的として GDB に直接入力された式をチェックすることはできます。作業言語の場合と同様に、GDB が自動的にチェックを行うか否かを、ユーザ・プログラムのソース言語によって決定することもできます。サポートされている言語のデフォルトの設定については、セクション 9.4 [サポートされる言語], ページ 84 を参照してください。

### 9.3.1 型チェックの概要

いくつかの言語、例えば Modula-2 などは、強く型付けされています。これは、演算子や関数への引数は正しい型でなくてはならず、そうでない場合にはエラーが発生するということを意味しています。このようなチェックは、型の不一致のエラーが実行時に問題を発生させるのを防いでくれます。例えば、 $1 + 2$  は

$$1 + 2 \Rightarrow 3$$

となりますが、 $1 + 2.3$  は

error  $1 + 2.3$

のようにエラーになります。第 2 の例がエラーになるのは、CARDINAL 型の 1 は REAL 型の 2.3 と型の互換性がないからです。

GDB コマンドの中で使われる式については、ユーザが GDB の型チェック機能に対して、以下のような指示を出すことができます。

- チェックを行わない
- あらゆる不一致をエラーとして扱い、式を破棄する
- 型の不一致が発生したときには警告メッセージを出力するだけで、式の評価を実行する

最後の指示が選択された場合、GDB は上記の第 2 の ( エラー ) 例のような式でも評価しますが、その際には警告メッセージを出力します。

型チェックをしないよう指示した場合でも、型に関係のある原因によって GDB が式の評価ができなくなる場合があります。例えば、GDB は `int` の値と `struct foo` の値を加算する方法を知りません。こうした特定の型エラーは、使用されている言語に起因するものではなく、この例のように、そもそも評価することが意味をなさないような式に起因するものです。

個々の言語は、それが型に関してどの程度厳密であるかを定義しています。例えば、Modula-2 と C はいずれも、算術演算子への引数としては数値を要求します。C では、列挙型とポインタは数値として表わすことができますので、これらは算術演算子への正当な引数となります。特定の言語に関する詳細については、セクション 9.4 [サポートされる言語]、ページ 84 を参照してください。

GDB は、型チェック機能を制御するためのコマンドをさらにいくつか提供しています。

`set check type auto`

カレントな作業言語に応じて、型チェックを実行する、または、実行しないよう設定します。個々の言語のデフォルトの設定については、セクション 9.4 [サポートされる言語]、ページ 84 を参照してください。

`set check type on`

`set check type off`

カレントな作業言語のデフォルトの設定を無視して、型チェックを実行する、または、実行しないよう設定します。その設定が言語のデフォルトと一致しない場合は、警告メッセージが出力されます。型チェックを実行するよう設定されているときの式の評価において型の不一致が発生した場合には、GDB はメッセージを出力して式の評価を終了させます。

`set check type warn`

型チェック機能に警告メッセージを出力させますが、式の評価自体は常に実行するよう試みさせます。式の評価は、他の原因のために不可能になる場合もあります。例えば、GDB には数値と構造体の加算はできません。

`show type` 型チェック機能のカレントな設定と、GDB がそれを自動的に設定しているか否かを表示します。

### 9.3.2 範囲チェックの概要

いくつかの言語（例えば、Modula-2）では、型の上限を超えるとエラーになります。このチェックは、実行時に行われます。このような範囲チェックは、計算結果がオーバーフローしたり、配列の要素へのアクセス時に使うインデックスが配列の上限を超えたりすることがないことを確実にすることによって、プログラムの正しさを確かなものにすることを意図したものです。

GDB コマンドの中で使う式については、範囲エラーの扱いを以下のいずれかにするよう GDB に指示することができます。

- 範囲エラーを無視する
- 範囲エラーを常にエラーとして扱い、式を破棄する
- 警告メッセージを出力するだけで、式を評価する

範囲エラーは、数値がオーバーフローした場合、配列インデックスの上限を超えた場合、どの型のメンバでもない定数が入力された場合に発生します。しかし、言語の中には、数値のオーバーフローをエラーとして扱わないものもあります。C 言語の多くの実装では、数学的演算によるオーバーフローは、結果の値を「一巡」させて小さな値にします。例えば、 $m$  が整数値の最大値、 $s$  が整数値の最小値とすると、

$$m + 1 \Rightarrow s$$

になります。これも個々の言語に固有な性質であり、場合によっては、個々のコンパイラやマシンに固有な性質であることもあります。特定の言語に関する詳細については、セクション 9.4 [サポートされる言語], ページ 84 を参照してください。

GDB は、範囲チェック機能を制御するためのコマンドをさらにいくつか提供しています。

`set check range auto`

カレントな作業言語に応じて、範囲チェックを実行する、または、実行しないよう設定します。個々の言語のデフォルトの設定については、セクション 9.4 [サポートされる言語], ページ 84 を参照してください。

`set check range on`

`set check range off`

カレントな作業言語のデフォルトの設定を無視して、範囲チェックを実行する、または、実行しないよう設定します。設定が言語のデフォルトとは異なる場合は、警告メッセージが出力されます。範囲チェックを実行する設定になっているときに範囲エラーが発生した場合は、メッセージが表示され、式の評価は終了させられます。

`set check range warn`

GDB の範囲チェック機能が範囲エラーを検出した場合、メッセージを出力し、式の評価を試みます。例えば、プロセスが、自分の所有していないメモリをアクセスした場合（多くの UNIX システムで典型的に見られる例です）など、他の理由によって式の評価が不可能な場合があります。

`show range`

範囲チェック機能のカレントな設定と、それが GDB によって自動的に設定されているか否かを表示します。

## 9.4 サポートされる言語

GDB は、C、C++、Fortran、Java、Chill、アセンブリ言語、Modula-2 をサポートしています。いくつかの GDB の機能は、使用されている言語にかかわらず、式の中で使用できます。GDB の @ 演算子、:: 演算子、および ‘{type}addr’ ( セクション 8.1 [式], ページ 63 参照 ) は、サポートされている任意の言語において使用することができます。

次節以降で、個々のソース言語が GDB によってどの程度までサポートされているのかを詳しく説明します。これらの節は、言語についてのチュートリアルやリファレンスとなることを意図したものではありません。むしろ、GDB の式解析機能が受け付ける式や、異なる言語における正しい入出力フォーマットのリファレンス・ガイドとしてのみ役に立つものです。個々の言語については良い書籍が数多く出ています。言語についてのリファレンスやチュートリアルが必要な場合は、これらの書籍を参照してください。

### 9.4.1 C/C++

C と C++ は密接に関連しているので、GDB の機能の多くは両方の言語に適用できます。このようなものについては、2 つの言語を一緒に議論します。

C++ のデバッグ機能は、C++ コンパイラと GDB によって協同で実装されています。したがって、C++ のコードを効率よくデバッグするには、GNU g++、HP ANSI C++ コンパイラ ( aCC ) などの、サポートされている C++ コンパイラで、C++ のプログラムをコンパイルしなければなりません。

GNU C++ を使用する場合、最高の結果を引き出すには、stabs デバッグ・フォーマットを使用してください。g++ のコマンドライン・オプション ‘-gstabs’、または、‘-gstabs+’ によって、このフォーマットを明示的に選択することができます。詳細については、セクション “Options for Debugging Your Program or GNU CC” in *Using GNU CC* の部分を参照してください。

#### 9.4.1.1 C/C++ 演算子

演算子は、特定の型の値に対して定義されなければなりません。例えば、+ は数値に対しては定義されていますが、構造体に対しては定義されていません。演算子は、型のグループに対して定義されることがよくあります。

C/C++ に対しては、以下の定義が有効です。

- 整数型には、任意の記憶クラス指定子を持つ int が含まれます。char、enum も整数型です。また、C++ の場合は、bool も整数型に含まれます。
- 浮動小数点型には、float、double、および、( ターゲット・プラットフォームにおいてサポートされていれば ) long double が含まれます。
- ポインタ型には、型 *type* に対して ( *type* \* ) により定義されるすべての型が含まれます。
- スカラ型には、上記のすべてが含まれます。

以下の演算子がサポートされています。これらは優先順位の低いものから順に並べられています。

|     |                                                                                                                                   |
|-----|-----------------------------------------------------------------------------------------------------------------------------------|
| ,   | カンマ、あるいは、順序付けの演算子です。カンマによって区切られたリストの中の式は、左から右の順で評価されます。最後に評価された式の結果が、式全体の評価結果になります。                                               |
| =   | 代入。代入された値が、代入式の値になります。スカラ型に対して定義されています。                                                                                           |
| op= | $a \text{ op} = b$ という形式の式において使用され、 $a = a \text{ op} b$ に変換されます。op= と = は、同一の優先順位を持ちます。op には、  ^ & << >> + - * / % の各演算子が使用できます。 |

|              |                                                                                                                                                                                                                           |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?:           | 3 項演算子です。 $a ? b : c$ は、 $a$ が真であれば $b$ 、偽であれば $c$ とみなすことができます。 $a$ は整数型でなければなりません。                                                                                                                                       |
|              | 論理 OR です。整数型に対して定義されています。                                                                                                                                                                                                 |
| &&           | 論理 AND です。整数型に対して定義されています。                                                                                                                                                                                                |
|              | ビットごとの OR です。整数型に対して定義されています。                                                                                                                                                                                             |
| ^            | ビットごとの排他的 OR です。整数型に対して定義されています。                                                                                                                                                                                          |
| &            | ビットごとの AND です。整数型に対して定義されています。                                                                                                                                                                                            |
| ==, !=       | 等価、および、不等価です。スカラ型に対して定義されています。これらの式の値は、偽のときはゼロであり、真のときはゼロ以外の値となります。                                                                                                                                                       |
| <, >, <=, >= | 未満、超過、以下、以上です。スカラ型に対して定義されています。これらの式の値は、偽のときはゼロであり、真のときはゼロ以外の値となります。                                                                                                                                                      |
| <<, >>       | 左シフト、右シフトです。整数型に対して定義されています。                                                                                                                                                                                              |
| @            | GDB の「人工配列」演算子です ( セクション 8.1 [式], ページ 63 参照 )。                                                                                                                                                                            |
| +, -         | 加算および減算です。整数型、浮動小数点型、ポインタ型に対して定義されています。                                                                                                                                                                                   |
| *, /, %      | 乗算、除算、剰余です。乗算と除算は、整数型と浮動小数点型に対して定義されています。剰余は、整数型に対して定義されています。                                                                                                                                                             |
| ++, --       | インクリメント、デクリメントです。変数の前にある場合は、式の中でその変数を使用される前に実行されます。変数の後ろにある場合は、変数の値が使用された後に実行されます。                                                                                                                                        |
| *            | ポインタの間接参照です。ポインタ型に対して定義されています。++ と同一の優先度を持ちます。                                                                                                                                                                            |
| &            | アドレス参照演算子です。変数に対して定義されています。++ と同一の優先順位を持ちます。<br><br>C++ のデバッグでは、C++ 言語そのものにおいては許されていないような '&' の使用法を、GDB は実装しています。C++ の ( '&ref' により宣言される ) 参照変数が格納されているアドレスを調べるのに、 '&(&ref)' (あるいは、もしそうしたいのであれば単に '&&ref' ) を使用することができます。 |
| -            | マイナス ( 負 ) です。整数型と浮動小数点型に対して定義されています。++ と同一の優先順位を持ちます。                                                                                                                                                                    |
| !            | 論理 NOT です。整数型に対して定義されています。++ と同一の優先順位を持ちます。                                                                                                                                                                               |
| ~            | ビットごとの NOT ( 補数 ) 演算子です。整数型に対して定義されています。++ と同一の優先順位を持ちます。                                                                                                                                                                 |
| ., ->        | 構造体のメンバ、ポインタの指す構造体のメンバをそれぞれ指定する演算子です。便宜上、GDB は両者を同一のものとして扱い、格納されている型情報をもとに、ポインタによる間接参照の必要性を判断します。構造体 ( struct ) および共用体 ( union ) に対して定義されています。                                                                            |
| .*, ->*      | メンバを指すポインタの間接参照です。                                                                                                                                                                                                        |

- [ ]        配列のインデックスです。a[i] は、\*(a+i)として定義されています。->と同一の優先順位を持ちます。
- ()        関数のパラメータ・リストです。->と同一の優先順位を持ちます。
- ::        C++のスコープ解決演算子です。構造体( struct )、共用体( union )、クラス( class )に対して定義されています。
- ::        2重コロンはまた、GDBのスコープ演算子も表わします( セクション 8.1 [式], ページ 63 参照 )。上記の::と同一の優先順位を持ちます。

ユーザ・コードの中で演算子が再定義されていると、GDB は通常、事前定義されている意味ではなく、再定義された意味において、その演算子を実行することを試みます。

#### 9.4.1.2 C/C++定数

GDB では、以下のような方法によって、C/C++の定数を表わすことができます。

- 整数型定数は、数字の連続したものです。8進数定数は、先頭の‘0’ (ゼロ)により指定されます。16進数定数は、先頭の‘0x’または‘0X’により指定されます。定数は、文字‘l’ (アルファベット小文字の「エル」)により終わることもあります。この場合、定数が long 型の値として扱われるべきことを意味します。
- 浮動小数点型定数は、連続した数字、その後ろに小数点、さらにその後ろに数字という形式です。場合によっては、最後に指数部が付くこともあります。指数部は、‘e[+|-]nnn’という形式を取ります。ここで、nnn は連続した数字です。‘+’は正の指数を示す記号で、必ずしも必要ではありません。浮動小数点型定数の末尾が、文字‘f’または‘F’となることもあります。この場合、定数が (デフォルトの double 型ではなく) float 型の値として扱われるべきことを意味します。また、文字‘l’または‘L’で終わる場合は、long double 型定数であることを意味します。
- 列挙型定数は、列挙識別子、またはそれに対応する整数値より構成されます。
- 文字型定数は、単一引用符( ‘ ’ )によって囲まれた単一の文字、あるいは、その文字に対応する序数 (通常は、ASCII 値) です。引用符の中の単一文字は、文字またはエスケープ・シーケンスによって表わすことができます。エスケープ・シーケンスには 2 つの表記方法があります。第 1 の形式は ‘\nnn’ で、nnn はその文字の序数を表わす 8 進数です。第 2 の形式は ‘\x’ で、‘x’ はあらかじめ定義された特別な文字です。例えば、‘\n’ は改行を表わします。
- 文字列型定数は、連続した文字定数が 2 重引用符( " " )で囲まれたものです。上記の正当な文字定数であれば、どれでも含むことができます。文字列の中に 2 重引用符を含める場合は、その前にバックスラッシュを置かなければなりません。したがって、例えば ‘"a\"b\"c"’ は、5 文字からなる文字列です。
- ポインタ型定数は、整数値です。定数へのポインタを、C の ‘&’ 演算子を使用して記述することができます。
- 配列定数は、括弧 ‘{’ と ‘}’ で囲まれ、カンマで区切られたリストです。例えば、‘{1,2,3}’ は 3 つの整数値を要素として持つ配列です。‘{{1,2},{3,4},{5,6}}’ は、3 × 2 の配列です。また、‘{&"hi", &"there", &"fred"}’ は 3 つのポインタを要素として持つ配列です。

#### 9.4.1.3 C++式

GDB が持っている、式を処理する機能は、C++のほとんどの式を解釈することができます。

注意：GDB は、適切なコンパイラが使用されている場合のみ、C++のコードをデバッグすることができます。典型的な例を挙げると、C++のデバッグでは、シンボ



ル・テーブルの中の追加的なデバッグ情報に依存するため、特別なサポートが必要になるということがあります。使用されるコンパイラが、a.out、MIPS ECOFF、RS/6000 XCOFF、ELF を、シンボル・テーブルに対する stabs 拡張付きで生成することができるのであれば、以下に列挙する機能を使用することができます (GNU CC の場合は、'-gstabs' オプションを使用して明示的に stabs デバッグ拡張を要求することができます)。一方、オブジェクト・コードのフォーマットが、標準 COFF や ELF の DWARF である場合には、GDB の提供するほとんどの C++ サポートは機能しません。

1. メンバ関数の呼び出しが許されます。以下のような式を使用することができます。

```
count = aml->GetOriginal(x, y)
```

2. メンバ関数が (選択されたスタック・フレームの中で) アクティブな場合、入力された式は、そのメンバ関数と同一の名前空間を利用することができます。すなわち、GDB は、C++ と同様の規則にしたがって、クラス・インスタンスへのポインタ this への暗黙の参照を許します。
3. オーバーロードされた関数を呼び出すことができます。GDB は、正しい定義の関数呼び出しを決定します。ただし、これには制限があります。GDB は、ユーザ定義の型に関連する変換、コンストラクタの呼び出し、プログラムの中に存在しないテンプレートのインスタンス生成を必要とするオーバーロードの解決を実行しません。さらに、省略記号を使用した引数リストやデフォルト引数を処理することもできません。

整数型の変換や拡張、浮動小数点拡張、算術変換、ポインタ変換、クラス・オブジェクトのベース・クラスへの変換、および、関数やポインタ配列などの標準的な変換は実行されます。関数引数の数は、正確に一致していなければなりません。

オーバーロードの解決は、set overload-resolution off が指定されていない限り、常に実行されます。セクション 9.4.1.7 [C++用の GDB 機能], ページ 88 を参照してください。次の例に示すように、あるオーバーロードされた関数を呼び出すために明示的な関数シグネチャを使用するためには、set overload-resolution off を指定しなければなりません。

```
p 'foo(char,int)' ('x', 13)
```

GDB のコマンド補完機能を利用すれば、このようなことは簡単になります。セクション 3.2 [コマンド名の補完], ページ 15 を参照してください。

4. GDB は、C++ の参照変数として宣言された変数を理解します。C++ のソース・コードで参照変数を使用するのと同じ方法で、参照変数を式の中で使用することができます。参照変数は自動的に間接参照されます。

GDB がフレームを表示する際に表示されるパラメータ一覧の中では、参照変数の値は (他の変数とは異なり) 表示されません。これにより、表示が雑然となることを回避できます。というのは、参照変数は大きい構造体に対して使用されることが多いからです。参照変数のアドレスは、'set print address off' を指定しない限り、常に表示されます。

5. GDB は C++ の名前解決演算子 :: をサポートしています。プログラム中と同様に、式の中でこれを使用することができます。あるスコープが別のスコープの中で定義されることがありえるため、必要であれば :: を繰り返し使用することができます。例えば、'scope1::scope2::name' という具合です。GDB はまた、C および C++ のデバッグにおいて、ソース・ファイルを指定することで名前のスコープを解決することを許します (セクション 8.2 [プログラム変数], ページ 64 参照)。

さらに、HP の C++ コンパイラを使用している場合、GDB は、仮想関数の正しい呼び出し、オブジェクトの仮想ベース・クラスの表示、ベース・サブオブジェクトの中の関数の呼び出し、オブジェクトのキャスト、ユーザ定義演算子の実行をサポートしています。

#### 9.4.1.4 C/C++のデフォルト

GDB が自動的に型チェックや範囲チェックの設定を行うことを許すと、作業言語が C や C++ に変更されるときにはいつも、それらの設定はデフォルトで off になります。これは、作業言語を選択したのがユーザであっても GDB であっても同様です。

GDB が自動的に言語の設定を行うことを許すと、GDB は、名前が `‘.c’`、`‘.C’`、`‘.cc’` などと終わるソース・ファイルを認識していて、これらのファイルからコンパイルされたコードの実行を開始するときに、作業言語を C または C++ に設定します。詳細については、セクション 9.1.3 [GDB によるソース言語の推定], ページ 80 を参照してください。

#### 9.4.1.5 C/C++の型チェックと範囲チェック

デフォルトでは、GDB が C や C++ の式を解析するときには、型チェックは行われません。しかし、ユーザが型チェックを有効にすると、GDB は以下の条件が成立するときに、2 つの変数の型が一致しているとみなします。

- 2 つの変数が構造を持ち、同一の構造体タグ、共用体タグ、または列挙型タグを持つ。
- 2 つの変数が同一の型名を持つ、あるいは、typedef によって同一の型として宣言されている型を持つ。

範囲チェックは、on に設定されている場合、数学的演算において実行されます。配列のインデックスは、それ自体は配列ではないポインタのインデックスとして使用されることが多いため、チェックされません。

#### 9.4.1.6 GDB と C

`set print union` コマンドと `show print union` コマンドは共用体型 (union) に適用されます。`‘on’` に設定されると、構造体 (struct) やクラス (class) の内部にある共用体 (union) はすべて表示されます。`‘on’` でない場合、それは `‘{...}’` と表示されます。

@オペレータは、ポインタとメモリ割り当て関数によって作られた動的配列のデバッグに役に立ちます。セクション 8.1 [式], ページ 63 を参照してください。

#### 9.4.1.7 C++用の GDB 機能

GDB のコマンドの中には、C++ を使用しているときに特に役に立つものがあり、また、C++ 専用に特に設計されたものがあります。以下に、その要約を示します。

`breakpoint menus`

名前がオーバーロードされている関数の内部にブレイクポイントを設定したい場合、関心のある関数定義を指定するのに、GDB のブレイクポイント・メニューが役に立ちます。セクション 5.1.8 [ブレイクポイント・メニュー], ページ 44 を参照してください。

`rbreak regex`

あるオーバーロードされたメンバ関数が、特別なクラスだけが持つメンバ関数というわけではない場合、そのメンバ関数にブレイクポイントを設定するのに、正規表現によるブレイクポイントの設定が役に立ちます。セクション 5.1.1 [ブレイクポイントの設定], ページ 32 を参照してください。

`catch throw`  
`catch catch`

C++の例外処理をデバッグするのに使用します。セクション 5.1.3 [キャッチポイントの設定], ページ 38 を参照してください。

`ptype typename`

型 *typename* に関して、継承関係などの情報を表示します。章 10 [シンボル・テーブルの検査], ページ 101 を参照してください。

`set print demangle`  
`show print demangle`  
`set print asm-demangle`  
`show print asm-demangle`

コードを C++のソースとして表示する場合と、逆アセンブル処理の結果を表示する場合に、C++のシンボルをソース形式で表示するか否かを制御します。セクション 8.7 [表示設定], ページ 70 を参照してください。

`set print object`  
`show print object`

オブジェクトの型を表示する際に、派生した ( 実際の ) 型と宣言された型のどちらを表示するかを選択します。セクション 8.7 [表示設定], ページ 70 を参照してください。

`set print vtbl`  
`show print vtbl`

仮想関数テーブルの表示形式を制御します。セクション 8.7 [表示設定], ページ 70 を参照してください。( `vtbl` コマンドは、HP ANSI C++コンパイラ ( `aCC` ) によってコンパイルされたプログラムに対しては機能しません。)

`set overload-resolution on`

C++の式の評価に際して、オーバーロード解決を有効化します。デフォルトは `on` です。GDB は、オーバーロードされた関数の引数を評価し、標準的な C++の変換規則 ( 詳細については、セクション 9.4.1.3 [C++ expressions], ページ 86 を参照 ) を利用して、引数の型がマッチするシグニチャを持つ関数を探します。マッチする関数を見つけることができない場合は、メッセージを出力します。

`set overload-resolution off`

C++の式の評価に際して、オーバーロード解決を無効化します。GDB は、オーバーロードされた関数のうちクラスのメンバ関数ではないものについては、引数が正しい型であるか否かにかかわらず、シンボル・テーブルの中で最初に見つかった、指定された名前を持つ関数を選択します。オーバーロードされた関数のうちクラスのメンバ関数でもあるものについては、引数の型が正確にマッチするシグニチャを持つ関数を探します。

オーバーロードされたシンボル名

オーバーロードされたシンボルを宣言するのに C++において使用されるのと同じの表記法を使用して、オーバーロードされたシンボル定義のうち、特定のものを指定することができます。単に *symbol* と入力するのではなく、*symbol(types)* と入力してください。GDB コマンドラインの単語補完機能を使用して、利用可能な選択肢を一覧表示させたり、型のリストを完結させたりすることができます。この機能

の使用方法の詳細については、セクション 3.2 [コマンド名の補完], ページ 15 を参照してください。

## 9.4.2 Modula-2

Modula-2 をサポートするために開発された GDB の拡張機能は、( 現在開発中の ) GNU Modula-2 コンパイラによって生成されたコードだけをサポートします。他の Modula-2 コンパイラは現在サポートされていません。他の Modula-2 コンパイラが生成した実行形式モジュールをデバッグしようとする、おそらく、GDB が実行モジュールのシンボル・テーブルを読み込もうとしたところでエラーになるでしょう。

### 9.4.2.1 Modula-2 演算子

演算子は、特定の型の値に対して定義されなければなりません。例えば、+ は数値に対して定義され、構造体に対しては定義されません。演算子は、型のグループに対して定義されることがよくあります。Modula-2 においては、以下の定義が有効です。

- 整数型は、INTEGER、CARDINAL、およびそのサブ範囲 ( subrange ) から成ります。
- 文字型は、CHAR とそのサブ範囲から成ります。
- 浮動小数点型は、REAL から成ります。
- ポインタ型は、POINTER TO *type* のように宣言された任意の型から成ります。
- スカラ型は、上記のすべての型から成ります。
- 集合型は、SET、BITSET から成ります。
- ブール型は、BOOLEAN から成ります。

以下の演算子がサポートされています。ここでは、優先順位の低いものから順に並べています。

|          |                                                                                                           |
|----------|-----------------------------------------------------------------------------------------------------------|
| ,        | 関数の引数の区切り記号、または、配列のインデックスの区切り記号です。                                                                        |
| :=       | 代入です。var := value の値は value です。                                                                           |
| <, >     | 未満、超過です。整数型、浮動小数点型、列挙型に対して定義されています。                                                                       |
| <=, >=   | 整数型、浮動小数点型、列挙型に対しては、以下、以上を表わします。集合型に対しては、集合の包含関係を表わします。< と同一の優先順位を持ちます。                                   |
| =, <>, # | スカラ型に対して定義されている等価および 2 種類の不等価です。< と同一の優先順位を持ちます。GDB スクリプトの中では、# がスクリプトのコメント記号でもあるため、不等価としては <> だけが使用可能です。 |
| IN       | 集合のメンバを表わします。集合型、およびそのメンバの型に対して定義されています。< と同一の優先順位を持ちます。                                                  |
| OR       | ブール型の OR ( disjunction ) です。ブール型に対して定義されています。                                                             |
| AND, &   | ブール型の AND ( conjunction ) です。ブール型に対して定義されています。                                                            |
| @        | GDB の「人工配列」演算子です ( セクション 8.1 [式], ページ 63 参照 )。                                                            |
| +, -     | 整数型、浮動小数点型に対しては、加算、減算を表わします。集合型に対しては、和集合 ( union )、差集合 ( difference ) を表わします。                             |
| *        | 整数型、浮動小数点型に対しては、乗算を表わします。集合型に対しては、積集合 ( intersection ) を表わします。                                            |

|                                                                                                                     |                                                                                      |
|---------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| /                                                                                                                   | 浮動小数点型に対しては、除算を表わします。集合型に対しては、対称的差集合 ( symmetric difference ) を表わします。*と同一の優先順位を持ちます。 |
| DIV, MOD                                                                                                            | 整数型の除算における商と剰余を表わします。整数型に対して定義されています。*と同一の優先順位を持ちます。                                 |
| -                                                                                                                   | マイナス ( 負 ) です。INTEGER、REAL型のデータに対して定義されています。                                         |
| ^                                                                                                                   | ポインタの間接参照です。ポインタ型に対して定義されています。                                                       |
| NOT                                                                                                                 | ブール型の NOT です。ブール型に対して定義されています。^と同一の優先順位を持ちます。                                        |
| .                                                                                                                   | RECORDフィールドの区切り記号です。RECORDデータに対して定義されます。^と同一の優先順位を持ちます。                              |
| []                                                                                                                  | 配列のインデックスを指定します。ARRAY型のデータに対して定義されています。^と同一の優先順位を持ちます。                               |
| ()                                                                                                                  | プロシージャの引数リストを指定します。PROCEDUREオブジェクトに対して定義されています。^と同一の優先順位を持ちます。                       |
| ::, .                                                                                                               | GDB および Modula-2 のスコープ指定演算子です。                                                       |
| 注意: 集合、および集合に対する操作は、まだサポートされていません。このため、GDB は IN演算子、あるいは、集合に対して+、-、*、/、=、<>、#、<=、>= のいずれかの演算子が使用された場合、これをエラーとして扱います。 |                                                                                      |

#### 9.4.2.2 組み込み関数と組み込みプロシージャ

Modula-2 では、いくつかの組み込みプロシージャ、組み込み関数が使用できます。これらの説明にあたり、以下のメタ変数を使用します。

|          |                                                                                                                                        |
|----------|----------------------------------------------------------------------------------------------------------------------------------------|
| <i>a</i> | ARRAY型の変数を表わします。                                                                                                                       |
| <i>c</i> | CHAR型の定数または変数を表わします。                                                                                                                   |
| <i>i</i> | 整数型の変数または定数を表わします。                                                                                                                     |
| <i>m</i> | 集合に属する識別子を表わします。通常、同一関数の中でメタ変数 <i>s</i> とともに使用されます。 <i>s</i> の型は、SET OF <i>mtype</i> でなければなりません ( ここでの <i>mtype</i> は <i>m</i> の型です )。 |
| <i>n</i> | 整数型または浮動小数点型の、変数または定数を表わします。                                                                                                           |
| <i>r</i> | 浮動小数点型の変数または定数を表わします。                                                                                                                  |
| <i>t</i> | 型を表わします。                                                                                                                               |
| <i>v</i> | 変数を表わします。                                                                                                                              |
| <i>x</i> | 多くの型の中の 1 つの型の、変数または定数を表わします。詳細については、関数の説明の部分参照してください。                                                                                 |

また、すべての Modula-2 の組み込みプロシージャは、以下に説明する値を返します。

|                 |                                                                          |
|-----------------|--------------------------------------------------------------------------|
| ABS( <i>n</i> ) | 値 <i>n</i> の絶対値を返します。                                                    |
| CAP( <i>c</i> ) | <i>c</i> が小文字であれば、それを大文字にして返します。 <i>c</i> が小文字でなければ、 <i>c</i> をそのまま返します。 |

|                             |                                                                                                                   |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------|
| CHR( <i>i</i> )             | 序数が <i>i</i> である文字を返します。                                                                                          |
| DEC( <i>v</i> )             | 変数 <i>v</i> の値から 1 を引きます。新しい値を返します。                                                                               |
| DEC( <i>v</i> , <i>i</i> )  | 変数 <i>v</i> の値から <i>i</i> で示される値を引きます。新しい値を返します。                                                                  |
| EXCL( <i>m</i> , <i>s</i> ) | 集合 <i>s</i> から要素 <i>m</i> を取り除きます。新しい集合を返します。                                                                     |
| FLOAT( <i>i</i> )           | 整数値 <i>i</i> に等しい浮動小数点値を返します。                                                                                     |
| HIGH( <i>a</i> )            | 配列 <i>a</i> の最後の要素のインデックスを返します。                                                                                   |
| INC( <i>v</i> )             | 変数 <i>v</i> の値に 1 を加えます。新しい値を返します。                                                                                |
| INC( <i>v</i> , <i>i</i> )  | 変数 <i>v</i> の値に <i>i</i> で示される値を加えます。新しい値を返します。                                                                   |
| INCL( <i>m</i> , <i>s</i> ) | 集合 <i>s</i> に要素 <i>m</i> が存在しない場合、要素 <i>m</i> を追加します。新しい集合を返します。                                                  |
| MAX( <i>t</i> )             | 型 <i>t</i> の最大値を返します。                                                                                             |
| MIN( <i>t</i> )             | 型 <i>t</i> の最小値を返します。                                                                                             |
| ODD( <i>i</i> )             | <i>i</i> が奇数であればブール型の TRUE を返します。                                                                                 |
| ORD( <i>x</i> )             | 引数の序数値を返します。例えば、文字の序数値は、( ASCII 文字セットをサポートするマシン上では ) その ASCII 値です。ここで <i>x</i> は、整数型、文字型、列挙型のような順序を持つ型でなければなりません。 |
| SIZE( <i>x</i> )            | 引数のサイズを返します。 <i>x</i> は変数または型のいずれかです。                                                                             |
| TRUNC( <i>r</i> )           | <i>r</i> の整数部を返します。                                                                                               |
| VAL( <i>t</i> , <i>i</i> )  | 型 <i>t</i> のメンバのうち、その序数値が <i>i</i> であるものを返します。                                                                    |

注意：集合、および集合に対する操作はまだサポートされていません。したがって、INCL プロシージャ、EXCL プロシージャを使用すると、GDB はエラーとして扱います。

### 9.4.2.3 定数

GDB では、Modula-2 の定数を以下のような方法で表現することができます。

- 整数型の定数は、単に数字が連続したものです。式の中で使用された場合、定数は、式の他の部分と互換性のある型を持つものとみなされます。16 進数の整数は末尾に 'H' を付加することで、また、8 進数の整数は末尾に 'B' を付加することで指定されます。
- 浮動小数点型の定数は、連続した数字、その後ろに小数点、さらにその後ろに連続した数字が続くものです。場合によっては、この後ろに指数部を指定することができます。指数部の形式は 'E[+|-]nnn' で、'[+|-]nnn' の部分で希望する指数を指定します。浮動小数点型定数のすべての数字は、有効な 10 進数値でなければなりません。
- 文字型定数は、単一引用符 ( ' ) または 2 重引用符 ( " ) で囲まれた単一文字より成ります。文字型定数は、その文字の序数値 ( 通常は ASCII 値 ) の後ろに 'C' を付加することで表現することもできます。
- 文字列型定数は、単一引用符 ( ' ) または 2 重引用符 ( " ) で囲まれた連続する文字から成ります。C 言語のスタイルでのエスケープ・シーケンスも使用できます。エスケープ・シーケンスに関する簡単な説明については、セクション 9.4.1.2 [C/C++ 定数], ページ 86 を参照してください。

- 列挙型定数は、列挙識別子から成ります。
- ブール型定数は、識別子 TRUE および FALSE から成ります。
- ポインタ型定数は、整数値だけから成ります。
- 集合型定数は、まだサポートされていません。

#### 9.4.2.4 Modula-2 デフォルト

型チェックと範囲チェックが GDB により自動的に設定される場合、作業言語が Modula-2 に変わるたびに、それらはデフォルトで on に設定されます。これは、作業言語を選択したのがユーザであろうと GDB であろうと同様です。

GDB に自動的に言語を設定させると、ファイル名の末尾が '.mod' であるファイルからコンパイルされたコードに入るたびに、作業言語は Modula-2 に設定されます。詳細については、セクション 9.1.3 [GDB によるソース言語の推定], ページ 80 を参照してください。

#### 9.4.2.5 標準 Modula-2 との差異

Modula-2 プログラムのデバッグを容易にするために若干修正が施されています。これは主に、型に対する厳密性を緩めることによって実現されています。

- 標準 Modula-2 とは異なり、ポインタ型定数は整数値から作成することができます。これにより、デバッグ中にポインタ変数の値を変更することができますようになります (標準 Modula-2 では、ポインタ変数に格納されている実際のアドレスを知ることはできません。ポインタ変数内のアドレスは、他のポインタ変数、または、ポインタを返す式を直接的に代入することによってのみ修正することができます)。
- 表示不可の文字を表わすのに、C 言語のエスケープ・シーケンスを文字列や文字において使用することができます。GDB はこれらのエスケープ・シーケンスを埋め込んだまま文字列を表示します。表示不可の単一文字は、'CHR(*nnn*)' という形式で表示されます。
- 代入演算子 (:=) は、右側の引数の値を返します。
- すべての組み込みプロシージャは、引数を修正し、さらにそれを返します。

#### 9.4.2.6 Modula-2 の型チェックと範囲チェック

注意: GDB は現在のところ、型チェック、範囲チェックをまだ実装していません。

GDB は、以下のいずれかの条件が成立するとき、2 つの Modula-2 変数の型が等しいとみなします。

- 2 つの型が、TYPE *t1* = *t2* 文によって等しいと宣言されている型である。
- 2 つの型が同一行において宣言されている (注: これは GNU Modula-2 コンパイラにおいては正しいのですが、他のコンパイラにおいては正しくない可能性があります)。

型チェックが有効である限り、等しくない型の変数を組み合わせようとする試みはすべてエラーとなります。

範囲チェックは、すべての数学的オペレーション、代入、配列のインデックス境界、および、すべての組み込み関数、組み込みプロシージャにおいて実行されます。

#### 9.4.2.7 スコープ演算子 :: と .

Modula-2 のスコープ演算子 (.) と GDB のスコープ演算子 (::) との間には 2、3 の微妙な相違点があります。この 2 つは似た構文を持っています。

```
module . id
scope :: id
```

ここで、*scope* はモジュール名またはプロシージャ名、*module* はモジュール名、*id* はユーザ・プログラムの中で宣言された任意の（異なるモジュール以外の）識別子です。

:: 演算子を使用すると、GDB は *scope* によって指定されたスコープにおいて識別子 *id* を探します。指定されたスコープにおいてそれを見つけないと、GDB は *scope* によって指定されたスコープを包含するすべてのスコープを探します。

. 演算子を使用すると、GDB はカレントなスコープにおいて、*module* によって指定された定義モジュールから取り込まれた、*id* によって指定される識別子を探します。この演算子では、識別子 *id* が定義モジュール *module* から取り込まれていない場合や *module* において *id* が識別子でない場合は、エラーになります。

#### 9.4.2.8 GDB と Modula-2

GDB コマンドの中には、Modula-2 プログラムのデバッグにはほとんど役に立たないものがあります。set print、show print の 5 つのサブ・コマンド ‘vtbl’、‘demangle’、‘asm-demangle’、‘object’、‘union’ は C/C++ にのみ適用されます。最初の 4 つは C++ に適用され、最後の 1 つは C の共用体 (union) に適用されます。これらは、Modula-2 において直接類似するものが存在しません。

@ 演算子 ( セクション 8.1 [式], ページ 63 参照 ) は、どの言語においても使用することができますが、Modula-2 においてはあまり役に立ちません。この演算子は、動的配列のデバッグを支援することを目的とするものですが、C/C++ では作成できる動的配列は、Modula-2 では作成できません。しかし、整数値定数によってアドレスを指定することができるので、‘{type}adrexpr’ は役に立ちます。

GDB スクリプトの中では、Modula-2 の不等価演算子 # はコメントの開始記号として解釈されます。代わりに <> を使用してください。

#### 9.4.3 Chill

Chill 言語をサポートするために GDB に加えられた拡張機能は、GNU Chill コンパイラによる出力しかサポートしていません。他の Chill コンパイラは現時点ではサポートされていません。他のコンパイラが生成した実行ファイルをデバッグしようとしても、GDB が実行ファイルのシンボル・テーブルを読み込むところでエラーになる可能性が高いでしょう。

このセクションでは、Chill に関連するトピックを取り上げます。また、これらのトピックをサポートする GDB の機能についても取り上げます。

##### 9.4.3.1 モードの表示方法

GDB が提供している Chill のデータ型 ( モード ) のサポートは、GNU Chill コンパイラの持つ機能に直接的な関係を持っているため、Chill 言語の標準仕様から若干逸脱しています。提供されているモードには以下のものがあります。

離散モード ( *Discrete Mode* ) :

- 整数モード ( *Integer Mode* )。これは BYTE, UBYTE, INT, UINT, LONG, ULONG によって事前定義されています。



- ブール・モード ( *Boolean Mode* )。これは BOOLによって事前定義されています。
- 文字モード ( *Character Mode* )。これは CHARによって事前定義されています。
- 集合モード ( *Set Mode* )。これはキーワード SETによって表示されます。

```
(gdb) ptype x
type = SET (karli = 10, susi = 20, fritzi = 100)
```

数が不定の集合型の場合、集合要素の値は省略されます。

- 範囲モード ( *Range Mode* )。これは以下によって表示されます。  

```
type = <basemode>(<lower bound> : <upper bound>)
```

 ここで、<lower bound>, <upper bound>は任意の離散リテラル式です(例えば、集合要素名)

パワーセット・モード ( *Powerset Mode* ) :

パワーセット・モード ( *Powerset Mode* )は、キーワード POWERSET、および、その後ろに続くパワーセットのメンバ・モードによって表示されます。メンバ・モードは任意の離散モードです。

```
(gdb) ptype x
type = POWERSET SET (egon, hugo, otto)
```

参照モード ( *Reference Mode* ) :

- 結合参照モード ( *Bound Reference Mode* )。これは、キーワード REF、および、その後ろに続く、参照が結合されているモード名によって表示されます。
- 非結合参照モード ( *Free Reference Mode* )。これはキーワード PTRによって表示されます。

プロシージャ・モード ( *Procedure Mode* )

プロシージャ・モード ( *Procedure Mode* )。これは、type = PROC(<parameter list>) <return mode> EXCEPTIONS (<exception list>) によって表示されます。<parameter list>は、パラメータ・モード ( *Parameter Mode* ) のリストです。<return mode>は、プロシージャに戻り値がある場合にそのモードを示します。<exception list>は、そのプロシージャの中で発生する可能性のあるすべての例外のリストです。

同期モード ( *Synchronization Mode* ) :

- イベント・モード ( *Event Mode* )。これは以下によって表示されます。

```
EVENT (<event length>)
```

(<event length>)は必須ではありません。

- バッファ・モード ( *Buffer Mode* )

```
BUFFER (<buffer length>)<buffer element mode>
```

(<buffer length>)は必須ではありません。

タイミング・モード ( *Timing Mode* ) :

- 時間範囲モード ( *Duration Mode* )。これは DURATIONによって事前定義されています。
- 絶対時刻モード ( *Absolute Time Mode* )。これは TIMEによって事前定義されています。

リアル・モード ( *Real Mode* ) :

リアル・モード ( *Real Mode* ) は、REAL と LONG\_REAL によって事前定義されています。

文字列モード ( *String Mode* ) :

- 文字列モード ( *Character String Mode* )。これは、  
CHARS(<string length>)  
によって表示されます。文字列モードが可変モード ( *varying mode* ) の場合は、この後ろにキーワード VARYING が続きます。
- ビット列モード ( *Bit String Mode* )。これは  
BOOLS(<string  
length>)  
によって表示されます。

配列モード ( *Array Mode* ) :

配列モード ( *Array Mode* ) は、キーワード ARRAY(<range>)、および、その後ろに続く要素モード ( *element mode* ) (これがまた配列モードである可能性があります) によって表示されます。

```
(gdb) ptype x
type = ARRAY (1:42)
      ARRAY (1:20)
        SET (karli = 10, susi = 20, fritzi = 100)
```

構造体モード ( *Structure Mode* )

構造体モード ( *Structure Mode* ) は、キーワード STRUCT(<field list>) によって表示されます。<field list> は、構造体フィールドの名前とモードから構成されます。可変構造体では、フィールド・リストの中にキーワード CASE <field> OF <variant fields> ESAC があります。現バージョンの GNU Chill コンパイラはタグの処理を実装していない (可変フィールドの実行時チェックは行われず、したがって、デバッグ情報も生成されません) ので、出力の中には常にすべての可変フィールドが表示されます。

```
(gdb) ptype str
type = STRUCT (
  as x,
  bs x,
  CASE bs OF
    (karli):
      cs a
    (ott):
      ds x
  ESAC
)
```

#### 9.4.3.2 ロケーションとそのアクセス

Chill におけるロケーション ( *location* ) とは、値を保持することのできるオブジェクトです。ロケーションの値は、一般的にはそのロケーションの (宣言された) 名前によってアクセスされます。出力は、Chill プログラムの値の仕様に準拠しています。値の指定方法は次のセクション (セクション 9.4.3.3 [Values and their Operations], ページ 97) のトピックとなります。

擬似ロケーション (pseudo-location) である RESULT (あるいは result) は、現在アクティブなプロシージャの戻り値を表示したり変更したりするのに使用することができます。

```
set result := EXPR
```

これは、( GDB の中では利用できない ) Chill のアクション RESULT EXPR と同じことを行います。参照モードのロケーションの値は、非結合参照モードの場合は PTR(<hex value>) によって、結合参照モードの場合は (REF <reference mode>) (<hex-value>) によって、それぞれ表示されます。<hex value> は、参照の指すアドレスを表わします。このポインタによって参照されるロケーションの値にアクセスするには、間接参照オペレータ ‘->’ を使ってください。

プロシージャ・モードのロケーションの値は、

```
{ PROC
  (<argument modes> ) <return mode> } <address> <name of procedure
  location>
```

によって表示されます。<argument modes> は、そのプロシージャのパラメータ指定にしたがったモードのリストです。<address> は、エントリ・ポイントのアドレスを示します。

文字列モードの値、配列モードの値、構造体モードの値などの下位構造 (例えば、配列スライスや構造体ロケーションのフィールド) は、次のセクション 9.4.3.3 [Values and their Operations], ページ 97 において説明される特定のオペレーションを使用してアクセスされます。

あるロケーションの値は、ロケーション変換を使用することによって、異なるモードを持つものとして解釈することができます。このモード変換は、<mode name>(<location>) のように記述されます。ユーザは、( 変換前と変換後の ) モードのサイズが等しいことを考慮しなければなりません。等しくない場合はエラーが発生します。さらに、変換後のモードに対してロケーションの範囲チェックは行われないため、変換結果が非常に分りにくいものになる可能性があります。

```
(gdb) print int (s(3 up 4)) XXX T0 be filled in !! XXX
```

### 9.4.3.3 値と操作

ロケーションの変更、複雑な構造体のより詳細な調査、大量のデータからの関連情報の抽出を行うのに値 (Value) が使用されます。このような調査を可能にするいくつかの (モード依存の) オペレーションが定義されています。これらのオペレーションは、定数値だけではなくロケーションに対しても適用可能です。このことが、複雑な構造体をデバッグする際に大変役に立つことがあります。コマンドラインの解析 (例えば、式の評価など) の際に、GDB はロケーション名をそのロケーションの背後にある値として取り扱います。

このセクションでは、値の指定方法と、個々の値の種類に対して正当に使用可能なオペレーションについて説明します。

#### リテラル値 (Literal Value)

リテラル値 (Literal Value) の指定方法は、GNU Chill プログラムにおける指定方法と同一です。詳細な仕様については、GNU Chill implementation Manual の 1.5 節を参照してください。

#### タプル値 (Tuple Value)

タプルは <mode name>[<tuple>] によって指定されます。<mode name> は、タプルのモードがあいまいでなければ省略可能です。この一義性 (あいまいさのなさ) は評価される式のコンテキストに由来します。<tuple> は以下のいずれかです。

- パワーセット・タプル (Powerset Tuple)
- 配列タプル (Array Tuple)

- 構造体タプル ( *Structure Tuple* )

パワーセット・タプル ( *Powerset Tuple* ), 配列タプル ( *Array Tuple* ), 構造体タプル ( *Structure Tuple* ) の指定方法は、Chill プログラムにおける指定方法と同一です。z.200/88 の 5.2.5 節を参照してください。

#### 文字列要素値 ( *String Element Value* )

文字列要素値 ( *String Element Value* ) は以下によって指定されます。

`<string value>(<index>)`

`<index>` は整数型の式です。これは、文字列の中の `<index>` で示されるインデックス位置の文字に等しい文字値を返します。

#### 文字列スライス値 ( *String Slice Value* )

文字列スライス値は `<string value>(<slice spec>)` によって指定されます。`<slice spec>` は、ある範囲の整数式、または、`<start expr> up <size>` による指定のいずれか一方です。`<size>` は、スライスの中に含まれる要素の数を表わします。返される値は文字列値であり、これは指定された文字列の一部分です。

#### 配列要素値 ( *Array Element Value* )

配列要素値 ( *Array Element Value* ) は `<array value>(<expr>)` によって指定され、指定された配列のモードにおける配列要素の値を返します。

#### 配列スライス値 ( *Array Slice Values* )

配列スライスは `<array value>(<slice spec>)` によって指定されます。`<slice spec>` は、式によって指定された範囲、または、`<start expr> up <size>` によって指定された範囲のいずれか一方です。`<size>` は、スライスの中に含まれる配列要素の数を表わします。返される値は、指定された配列の一部分である配列値です。

#### 構造体フィールド値 ( *Structure Field Value* )

構造体フィールド値 ( *Structure Field Value* ) は、`<structure value>.<field name>` によって取られます。`<field name>` は、構造体のモード定義において指定されたフィールドの名前を表わします。返される値のモードは、構造体定義におけるこのモード定義に対応します。

#### プロシージャ・コール値 ( *Procedure Call Value* )

プロシージャ・コール値 ( *Procedure Call Value* ) は、プロシージャの戻り値から取られます。<sup>1</sup>

時間範囲モード ( *duration mode* ) のロケーションの値は、ULONG リテラルによって表現されます。

絶対時刻モード ( *time mode* ) のロケーションの値は、以下のように示されます。

`TIME(<secs>:<nsecs>)`

#### ゼロ・アディック演算子値 ( *Zero-adic Operator Value* )

ゼロ・アディック演算子値 ( *Zero-adic Operator Value* ) は、現在アクティブなプロセスのインスタンス値から取られます。

<sup>1</sup> 原注：プロシージャ・コールが例えば式の中で使われている場合は、このプロシージャの呼び出しにともなうすべての副作用が発生します。これは、注意せずに使うと混乱をもたらす可能性があります。

## 式値 (Expression Value)

式によって返される値はその式の評価結果です。エラー条件 (モード非互換性など) が存在すると、式の評価は中断され、対応するエラー・メッセージが出力されます。式は括弧で囲むことができます。これによって、括弧で囲まれた式の結果を利用する他の式よりも先に、この括弧で囲まれた式を評価させることができます。GDB では以下の演算子がサポートされています。

|                |                                                                                        |
|----------------|----------------------------------------------------------------------------------------|
| OR, ORIF, XOR  |                                                                                        |
| AND, ANDIF     |                                                                                        |
| NOT            | ブール・モードのオペランドに対して定義された論理演算子                                                            |
| =, /=          | すべてのモードに対して定義された等価演算子、非等価演算子                                                           |
| >, >=          |                                                                                        |
| <, <=          | 事前定義されたモードに対して定義された関係演算子                                                               |
| +, -           |                                                                                        |
| *, /, MOD, REM | 事前定義されたモードに対して定義された算術演算子                                                               |
| -              | 符号変更演算子                                                                                |
| //             | 文字列連結演算子                                                                               |
| ()             | 文字列反復演算子                                                                               |
| ->             | あるロケーションのアドレスを取る (->loc)、あるいは、ある参照ロケーションを間接参照する (loc->) という目的で使用するの<br>できる被参照ロケーション演算子 |
| OR, XOR        |                                                                                        |
| AND            |                                                                                        |
| NOT            | パワーセット演算子およびビット列演算子                                                                    |
| >, >=          |                                                                                        |
| <, <=          | パワーセット包含演算子                                                                            |
| IN             | メンバ演算子                                                                                 |

## 9.4.3.4 Chill の型チェックと範囲チェック

GDB は、2 つの Chill 変数モードが存在するときに、その 2 つのモードのサイズが等しいと、それらを同等であるとみなします。このルールは、より複雑なデータ型についても再帰的に適用されます。すなわち、すべての要素のモード (これが再び構造体、配列のような複雑なモードである可能性があります) が同一のサイズである場合に、複雑なモードは同等なものとして扱われます。

範囲チェックは、すべての数学的オペレーション、代入、配列のインデックス境界、すべての組み込みプロシージャに対して実行されます。

強い型チェックは、GDB の `set check strong` コマンドを使うことによって強制されます。これによって、(式、組み込みプロシージャなどの) Chill 構成物が使用されているすべてのオペレーションに対して、z.200 言語仕様に定義されている意味との関連において、強い型チェックと強い範囲チェックが強制されます。

すべてのチェックは、GDB の `set check off` コマンドによって無効にすることができます。

#### 9.4.3.5 Chill のデフォルト

型チェックと範囲チェックが GDB により自動的に設定される場合、作業言語が Chill に変わるたびに、それらはデフォルトで on に設定されます。これは、作業言語を選択したのがユーザであろうと GDB であろうと同様です。

GDB に自動的に言語を設定させると、ファイル名の末尾が `‘.ch’` であるファイルからコンパイルされたコードに入るたびに、作業言語は Chill に設定されます。詳細については、セクション 9.1.3 [GDB によるソース言語の推定], ページ 80 を参照してください。

## 10 シンボル・テーブルの検査

ここで説明するコマンドによって、ユーザ・プログラムの中で定義されているシンボル情報（変数名、関数名、型名）に関する問い合わせを行うことができます。この情報はユーザ・プログラムのテキストに固有のもので、プログラムの実行時に変わるものではありません。GDBはこの情報を、ユーザ・プログラムのシンボル・テーブルの中、または、GDB 起動時に指定されたファイル（セクション 2.1.1 [ファイルの選択], ページ 10 参照）の中で見つけるか、ファイル管理コマンド（セクション 12.1 [ファイルを指定するコマンド], ページ 109 参照）の実行によって見つけます。

ときには、参照する必要のあるシンボルの中に、GDB が通常は単語の区切り文字として扱う文字が含まれていることがあるかもしれません。特に多いのが、他のソース・ファイルの中の静的変数を参照する場合です（セクション 8.2 [プログラム変数], ページ 64 参照）。ファイル名は、オブジェクト・ファイルの中にデバッグ・シンボルとして記録されていますが、GDB は通常、典型的なファイル名、例えば `'foo.c'` を解析して、3 つの単語 `'foo'`、`'.'`（ピリオド）、`'c'` であるとみなします。GDB が `'foo.c'` を単一のシンボルであると認識できるようにするには、それを単一引用符で囲みます。例えば、

```
p 'foo.c'::x
```

は、`x` の値をファイル `'foo.c'` のスコープの中で検索します。

`info address symbol`

`symbol` で指定されるシンボルのデータがどこに格納されているかを示します。レジスタ変数の場合は、それがどのレジスタに入っているかを示します。レジスタ変数ではないローカル変数の場合は、その変数が常に格納されている位置の、スタック・フレーム内におけるオフセット値を表示します。

`'print &symbol'` との相違に注意してください。`'print &symbol'` はレジスタ変数に対しては機能しませんし、スタック内のローカル変数に対して実行すると、その変数のカレントなインスタンスの存在するアドレスそのものが表示されます。

`what is expr`

式 `expr` のデータ型を表示します。`expr` は実際には評価されず、`expr` 内の副作用を持つ操作（例えば、代入や関数呼び出し）は実行されません。セクション 8.1 [式], ページ 63 を参照してください。

`what is`      値履歴の最後の値である `$` のデータ型を表示します。

`ptype typename`

データ型 `typename` の説明を表示します。`typename` は型の名前です。C で記述されたコードの場合は、`'class class-name'`、`'struct struct-tag'`、`'union union-tag'`、`'enum enum-tag'` という形式を取ることができます。

`ptype expr`

`ptype`      式 `expr` の型に関する説明を表示します。単に型の名前を表示するだけでなく、詳細な説明も表示するという点で、`ptype` は `what is` と異なります。

例えば、変数宣言

```
struct complex {double real; double imag;} v;
```

に対して、`what is`、`ptype` はそれぞれ以下のような出力をもたらします。

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

`whatis`と同様、引数なしで `ptype` を使用すると、値履歴の最後の値である \$ の型を参照することになります。

`info types regexp`

`info types`

その名前が `regexp` で指定される正規表現にマッチするすべての型 (あるいは、引数を指定しなければ、ユーザ・プログラム中のすべての型) の簡単な説明を表示します。個々の型の完全な名前は、それ自体が 1 つの完全な行であるものとみなしてマッチされます。したがって、`'i type value'` は、ユーザ・プログラムの中で、その名前が文字列 `value` を含むすべての型に関する情報を表示し、`'i type ^value$'` は、名前が `value` そのものである型に関する情報だけを表示します。

このコマンドは `ptype` とは 2 つの点で異なります。まず第 1 に `whatis` と同様、詳細な情報を表示しません。第 2 に、型が定義されているすべてのソース・ファイルを一覧表示します。

`info source`

カレントなソース・ファイル、すなわち、カレントな実行箇所を含む関数のソース・ファイルの、ファイル名とそれが記述された言語の名前を表示します。

`info sources`

ユーザ・プログラムのソース・ファイルのうち、デバッグ情報の存在するものすべての名前を、2 つの一覧にして表示します。2 つの一覧とは、シンボルが既に読み込まれたファイルの一覧と、後に必要なときにシンボルが読み込まれるファイルの一覧です。

`info functions`

すべての定義済み関数の名前とデータ型を表示します。

`info functions regexp`

その名前が `regexp` で指定される正規表現にマッチする部分を持つすべての定義済み関数の名前とデータ型を表示します。したがって、`'info fun step'` は、その名前が文字列 `step` を含むすべての関数を見つけ、`'info fun ^step'` は、名前が文字列 `step` で始まるすべての関数を見つけます。

`info variables`

関数の外部で宣言されているすべての変数 (つまり、ローカル変数を除く変数) の名前とデータ型を表示します。

`info variables regexp`

その名前が正規表現 `regexp` にマッチする部分を持つすべての (ローカル変数を除く) 変数の名前とデータ型を表示します。

いくつかのシステムにおいては、ユーザ・プログラムの停止・再起動を伴うことなく、そのユーザ・プログラムを構成する個々のオブジェクト・ファイルを更新することができます。例えば、VxWorks では、欠陥のあるオブジェクト・ファイルを



再コンパイルして、実行を継続することができます。このようなマシン上でプログラムを実行しているのであれば、自動的に再リンクされたモジュールのシンボルを GDB に再ロードさせることができます。

`set symbol-reloading on`

ある特定の名前を持つオブジェクト・ファイルが再検出されたときに、対応するソース・ファイルのシンボル定義を入れ替えます。

`set symbol-reloading off`

同じ名前を持つオブジェクト・ファイルを 2 回以上検出したときに、シンボル定義を入れ替えません。これがデフォルトの状態です。モジュールの自動再リンクを許しているシステム上でプログラムを実行しているのではない場合は、`symbol-reloading` の設定は `off` のままにするべきです。さもないと、(異なるディレクトリやライブラリの中にある) 同じ名前を持ついくつかのモジュールを含むような大きなプログラムをリンクする際に、GDB はシンボルを破棄してしまうかもしれません。

`show symbol-reloading`

`symbol-reloading` のカレントな設定 (`on` または `off`) を表示します。

`set opaque-type-resolution on`

オpaque (opaque) 型の解決を行うよう GDB に指示します。オpaque型とは、`struct`、`class`、または、`union` へのポインタとして宣言されている型—例えば、`struct MyType *`—であり、かつ、`struct MyType`<sup>1</sup> の完全な宣言が行われているソース・ファイルとは異なるソース・ファイルにおいて使用される型のことです。デフォルトは `on` です。

このサブコマンドの設定を変更しても、ファイルのシンボルが次にロードされるまでは効力を持ちません。

`set opaque-type-resolution off`

オpaque型の解決を行わないよう GDB に指示します。この場合、オpaque型は以下のように表示されます。

```
{<no data fields>}
```

`show opaque-type-resolution`

オpaque型の解決が行われるか否かを示します。

`maint print symbols filename`

`maint print psymbols filename`

`maint print msymbols filename`

デバッグ・シンボル・データのダンプをファイル `filename` の中に書き込みます。これらのコマンドは、GDB のシンボル読み込みコードをデバッグするのに使われています。デバッグ・データを持つシンボルだけがダンプに含まれます。‘`maint print symbols`’を使用すると、GDB は、完全な詳細情報を既に入手済みのすべてのシンボルをダンプに含めます。つまり、ファイル `filename` には、GDB がそのシンボルを読み込み済みのファイルに対応するシンボルが反映されます。info sources コ

<sup>1</sup> 訳注：その型 (ポインタ) の指す実体

マンドを使用することで、これらのファイルがどれであるかを知ることができます。代わりに `'maint print psymbols'` を使用すると、GDB が部分的にしか知らないシンボルに関する情報もダンプの中に含まれます。これは、GDB がざっと読みはしたものの、まだ完全には読み込んでいないファイルに定義されているシンボルに関する情報です。最後に `'maint print msymbols'` では、GDB が何らかのシンボル情報を読み込んだオブジェクト・ファイルから、最小限必要とされるシンボル情報がダンプされます。GDB がどのようにしてシンボルを読み込むかについては、セクション 12.1 [ファイルを指定するコマンド], ページ 109 ( の `symbol-file` の説明の部分 ) を参照してください。

## 11 実行処理の変更

ユーザ・プログラムの中に誤りのある箇所を見つけると、その明らかな誤りを訂正することで、その後の実行が正しく行われるかどうかを知りたくなるでしょう。GDB にはプログラムの実行に変化を与える機能があり、これを使って実験することで、その答を知ることができます。

例えば、変数やメモリ上のある箇所に新しい値を格納すること、ユーザ・プログラムにシグナルを送ること、ユーザ・プログラムを異なるアドレスで再起動すること、関数が完全に終了する前に呼び出し元に戻るなどが可能です。

### 11.1 変数への代入

ある変数の値を変更するには、代入式を評価します。セクション 8.1 [式], ページ 63 を参照してください。例えば、

```
print x=4
```

は、変数 `x` に値 4 を格納してから、その代入式の値 (すなわち 4) を表示します。サポートされている言語の演算子の詳細情報については、章 9 [異なる言語の使い方], ページ 79 を参照してください。

代入の結果を表示させることに関心がなければ、`print` コマンドの代わりに `set` コマンドを使用してください。実際のところ `set` コマンドは、式の値が表示されず、値履歴 (セクション 8.8 [値履歴], ページ 75 参照) にも入らないということを除けば、`print` コマンドと同等です。式は、その結果の入手だけを目的として評価されます。

`set` コマンドの引数となる文字列の先頭の部分が、`set` コマンドのサブ・コマンドの名前と一致してしまうような場合には、ただの `set` コマンドではなく `set variable` コマンドを使用してください。このコマンドは、サブ・コマンドを持たないという点を除けば、`set` コマンドと同等です。例えば、ユーザ・プログラムに `width` という変数がある場合、`'set width=13'` によってこの変数に値を設定しようとするとエラーになります。これは、GDB が `set width` というコマンドを持っているためです。

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

ここで不正な表現となっているのは、もちろん `'=47'` の部分です。プログラム内の変数 `width` に値を設定するには、以下のようにしてください。

```
(gdb) set var width=47
```

`set` コマンドは、プログラムの変数名と衝突する可能性のあるサブコマンドを多く持っているので、ただの `set` コマンドではなく、`set variable` コマンドを使用する方が良いでしょう。例えば、プログラムの中に `g` という名前の変数がある場合に、単に `'set g=4'` として新しい値をセットしようとすると、問題が発生します。これは、GDB が `set gnutarget` というコマンドを持っていて、その省略形が `set g` であるからです。

```

(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".

```

プログラム変数 *g* の値は変わらず、*g*nutarget にこっそりと不正な値をセットしてしまいました。変数 *g* に値をセットするためには、以下のようにします。

```
(gdb) set var g=4
```

GDB は、代入時の暗黙の型変換を C 言語よりも多くサポートしています。整数値を自由にポインタ型変数に格納できますし、その逆もできます。また、任意の構造体を、同じサイズの別の構造体、または、より小さいサイズの別の構造体に変換することができます。

メモリ上の任意の箇所に値を格納するには、指定されたアドレスにおいて指定された型の値を生成するために、`{...}` を使用します (セクション 8.1 [式], ページ 63 参照)。例えば `{int}0x83040` は、メモリ・アドレス `0x83040` を整数値として参照します (メモリ上における、ある特定のサイズと表現を示唆しています)。また、

```
set {int}0x83040 = 4
```

は、そのメモリ・アドレスに値 4 を格納します。

## 11.2 異なるアドレスにおける処理継続

通常、ユーザ・プログラムを継続実行するには、`continue` コマンドを使用して、停止した箇所から継続実行させます。以下のコマンドを使用することで、ユーザが選択したアドレスにおいて実行を継続させることができます。

`jump linespec`

*linespec* で指定される行において、実行を再開します。その行にブレイクポイントが設定されている場合には、実行は再びすぐに停止します。*linespec* の形式については、セクション 7.1 [ソース行の表示], ページ 57 を参照してください。一般的な慣例として、`jump` コマンドは、`tbreak` コマンドと組み合わせて使用されます。セクション 5.1.1 [ブレイクポイントの設定], ページ 32 を参照してください。

`jump` コマンドは、カレントなスタック・フレーム、スタック・ポインタ、メモリ内の任意の箇所の内容、プログラム・カウンタを除くレジスタの内容を変更しません。*linespec* で指定される行が、現在実行されている関数とは異なる関数の中にある場合、それら 2 つの関数が異なるパターンの引数やローカル変数を期待していると、奇妙な結果が発生するかもしれません。このため、指定された行が、現在実行されている関数の中にある場合、`jump` コマンドは実行の確認を求めてきます。しかし、

ユーザがプログラムのマシン言語によるコードを熟知していたとしても、奇妙な結果の発生することが予想されます。

`jump *address`

`address` で指定されるアドレスにある命令から実行を再開します。

多くのシステムでは、レジスタ `$pc` に新しい値を設定することで、`jump` コマンドとほとんど同等の効果を実現することができます。両者の違いは、レジスタ `$pc` に値を設定しただけでは、ユーザ・プログラムの実行は再開されないという点にあります。ユーザが実行を継続するときに、プログラムが実行を再開するであろうアドレスが変更されるだけです。例えば、

```
set $pc = 0x485
```

を実行すると、次に `continue` コマンドやステップ実行を行うコマンドが実行されるとき、ユーザ・プログラムが停止したアドレスにある命令ではなく、アドレス `0x485` にある命令から実行されることになります。セクション 5.2 [継続実行とステップ実行], ページ 45 を参照してください。`jump` コマンドが最も一般的に使用されるのは、既に実行されたプログラム部分を、さらに多くのブレイクポイントを設定した状態で再実行する場合でしょう。これにより、実行される処理の内容をさらに詳しく調べることができます。

### 11.3 ユーザ・プログラムへのシグナルの通知

`signal signal`

実行を停止した箇所からユーザ・プログラムを再開させますが、すぐに `signal` で指定されるシグナルを通知します。`signal` には、シグナルの名前または番号を指定できます。例えば、多くのシステムにおいて、`signal 2` と `signal SIGINT` はどちらも、割り込みシグナルを通知する方法です。

一方、`signal` が 0 であれば、シグナルを通知することなく実行を継続します。ユーザ・プログラムがシグナルのために停止し、通常であれば、`continue` コマンドによって実行を再開するとそのシグナルを検知してしまうような場合に便利です。`'signal 0'` を実行すると、プログラムはシグナルを受信することなく実行を再開します。

`signal` を実行した後、`[RET]` キーを押しても、繰り返し実行は行われません。

`signal` コマンドを実行することは、シェルから `kill` ユーティリティを実行するのと同じではありません。`kill` によってシグナルを送ると、GDB はシグナル処理テーブルによって何をすべきかを決定します (セクション 5.3 [シグナル], ページ 48 参照)。一方、`signal` コマンドは、ユーザ・プログラムに直接シグナルを渡します。

### 11.4 関数からの復帰

`return`

`return expression`

`return` コマンドによって、呼び出されている関数の実行をキャンセルすることができます。式 `expression` を引数に指定すると、その値が関数の戻り値として使用されます。

`return` を実行すると、GDB は選択されているスタック・フレーム (および、その下位にあるすべてのフレーム) を破棄します。破棄されたフレームは、実行を完結する前に復帰したのだと考

えればよいでしょう。戻り値を指定したいのであれば、その値を `return` への引数として渡してください。

このコマンドは、選択されているスタック・フレーム（セクション 6.3 [フレームの選択]、ページ 53 参照）および、その下位にあるすべてのフレームをポップして、もともと選択されていたフレームを呼び出したフレームを最下位のフレームにします。つまり、そのフレームが選択されることになります。指定された値は、関数から戻り値を返すのに使用されるレジスタに格納されます。

`return` コマンドは実行を再開しません。関数から復帰したと仮定した場合にその復帰直後にあるであろう状態で、プログラムを停止したままにします。これに対して、`finish` コマンド（セクション 5.2 [継続実行とステップ実行]、ページ 45 参照）は、選択されているスタック・フレームが自然に復帰するまで、実行を再開、継続します。

## 11.5 プログラム関数の呼び出し

`call expr` void型の戻り値を表示することなく、式 `expr` を評価します。

ユーザ・プログラムの中からある関数を呼び出したいが、void型の戻り値を出力させたくない場合、この `print` コマンドの変種を使用することができます。void型でない戻り値は表示され、値履歴に保存されます。

A29K では、ユーザが制御する変数 `call_scratch_address` によって、GDB がデバッグ対象の関数を呼び出すときに使用するスクラッチ領域が指定されます。通常はスクラッチ領域をスタック上に置きますが、この方法は命令空間とデータ空間を別々に持つシステム上では機能しないため、これが必要になります。

## 11.6 プログラムへのパッチ適用

デフォルトでは、GDB はユーザ・プログラムの実行コードを持つファイル（あるいは、コア・ファイル）を読み取りしかできない状態でオープンします。これにより、マシン・コードを誤って変更してしまうことを防ぐことができます。しかし、ユーザ・プログラムのバイナリに意図的にパッチを適用することもできなくなってしまいます。

バイナリにパッチを適用したいのであれば、`set write` コマンドによって明示的にそのことを指定することができます。例えば、内部的なデバッグ・フラグを立てたり、緊急の修正を行いたいということがあるでしょう。

```
set write on
set write off
```

‘`set write on`’を指定すると、GDB は実行ファイルやコア・ファイルを、読み取り、書き込みともに可能な状態でオープンします。‘`set write off`’（デフォルト）を指定すると、GDB はこれらのファイルを読み取りしかできない状態でオープンします。

既にファイルをロード済みの場合、`set write` の設定を変更後、その変更を反映させるためには、( `exec-file` コマンド、`core-file` コマンドを使用して ) そのファイルを再ロードしなければなりません。

```
show write
```

実行ファイル、コア・ファイルが、読み取りだけでなく書き込みもできる状態でオープンされる設定になっているか否かを表示します。

## 12 GDB ファイル

GDB はデバッグ対象となるプログラムのファイル名を知っている必要があります。これは、プログラムのシンボル・テーブルを読み込むためでもあり、また、プログラムを起動するためでもあります。過去に生成されたコア・ダンプをデバッグするには、GDB にコア・ダンプ・ファイルの名前を教えてやらなければなりません。

### 12.1 ファイルを指定するコマンド

実行ファイルやコア・ダンプ・ファイルの名前を指定したい場合があります。これは通常、GDB の起動コマンドへの引数を利用して起動時に行います( 章 2 [GDB の起動と終了], ページ 9 参照 )。ときには、GDB のセッション中に、異なるファイルに切り替える必要がでてくることがあります。あるいは、GDB を起動するときに、使いたいファイルの名前を指定するのを忘れたということもあるかもしれません。このような場合に、新しいファイルを指定する GDB コマンドが便利です。

`file filename`

`filename` で指定されるプログラムをデバッグ対象にします。そのプログラムは、シンボル情報とメモリ内容を獲得するために読み込まれます。また、ユーザが `run` コマンドを使用したときに実行されます。ユーザがディレクトリを指定せず、そのファイルが GDB の作業ディレクトリに見つからない場合、シェルが実行すべきプログラムを探すときと同様、GDB は、ファイルを探すべきディレクトリのリストとして環境変数 `PATH` の値を使用します。`path` コマンドによって、GDB、ユーザ・プログラムの両方について、この変数の値を変更することができます。

ファイルをメモリにマップすることのできるシステムでは、補助的なファイル '`filename.syms`' に、ファイル `filename` のシンボル・テーブル情報が格納されることがあります。このような場合、GDB は、'`filename.syms`' というファイルからシンボル・テーブルをメモリ上にマップすることで、起動に要する時間を短くします。詳細については、( 以下に説明する `file` コマンド、`symbol-file` コマンド、`add-symbol-file` コマンドを実行する際にコマンドライン上で使用可能な ) ファイル・オプションの '`-mapped`'、'`-readnow`' の説明を参照してください。

`file`      `file` コマンドを引数なしで実行すると、GDB は実行ファイル、シンボル・テーブルに関して保持している情報をすべて破棄します。

`exec-file [ filename ]`

実行するプログラムが `filename` で指定されるファイル内に存在する (ただし、シンボル・テーブルはそこには存在しない) ということを指定します。GDB は、必要であれば、ユーザ・プログラムの存在場所を見つけるために、環境変数 `PATH` を使用します。`filename` を指定しないと、実行ファイルに関して保持している情報を破棄するよう指示したことになります。

`symbol-file [ filename ]`

`filename` で指定されるファイルからシンボル・テーブル情報を読み込みます。必要な場合には `PATH` が検索されます。同一のファイルから、シンボル・テーブルと実行プログラムの両方を獲得する場合には、`file` コマンドを使用してください。

`symbol-file` を引数なしで実行すると、GDB がユーザ・プログラムのシンボル・テーブルに関して持っている情報は消去されます。

`symbol-file` コマンドが実行されると、それまで GDB が保持していたコンピュエンス変数、値履歴、すべてのブレイクポイント、自動表示式は破棄されます。その理由は、GDB が破棄した古いシンボル・テーブルのデータの一部であるシンボルやデータ型を記録する内部データへのポインタが、これらの情報の中に含まれているかもしれないからです。

`symbol-file` を一度実行した後に `(RET)` キーを押しても、`symbol-file` の実行は繰り返されません。

GDB は、`configure` によって特定の環境用に構成されると、その環境において生成される標準フォーマットのデバッグ情報を理解するようになります。GNU コンパイラを使うこともできますし、ローカルな環境の規約に従う他のコンパイラを使用することもできます。通常は、GNU コンパイラを使用しているときに最高の結果を引き出すことができます。例えば `gcc` を使用すると、最適化されたコードに対してデバッグ情報を生成することができます。

COFF を使用する古い SVR3 システムを除外すれば、ほとんどの種類のオブジェクト・ファイルでは、`symbol-file` コマンドを実行しても、通常は、ただちにシンボル・テーブルの全体が読み込まれるわけではありません。実際に存在するソース・ファイルとシンボルを知るために、シンボル・テーブルを素早く調べるだけです。詳細な情報は、後にそれが必要になったときに、一度に 1 ソース・ファイルずつ読み込まれます。

2 段階に分けて読み込むという手法は、GDB の起動時間の短縮を目的としています。ほとんどの場合、このような手法が採用されているということに気付くことはありません。せいぜい、特定のソース・ファイルに関するシンボル・テーブルの詳細が読み込まれている間、たまに停止するくらいです（もしそうしたいのであれば、`set verbose` コマンドを使うことによって、このようにして停止しているときにはメッセージを表示させることもできます。セクション 15.6 [オプションの警告およびメッセージ]、ページ 160 を参照してください）。

COFF については、まだこの 2 段階方式を実装していません。シンボル・テーブルが COFF フォーマットで格納されている場合、`symbol-file` コマンドはシンボル・テーブル・データの全体をただちに読み込みます。COFF の `stabs` 拡張フォーマット (`stabs-in-COFF`) では、デバッグ情報が実際には `stabs` フォーマットの内部に存在するため、2 段階方式が実装されていることに注意してください。

```
symbol-file filename [ -readnow ] [ -mapped ]
file filename [ -readnow ] [ -mapped ]
```

GDB が確実にシンボル・テーブル全体を保持しているようにしたいのであれば、シンボル・テーブル情報を読み込む任意のコマンド実行時に `-readnow` オプションを使用することで、2 段階によるシンボル・テーブル読み込み方式を使わないようにさせることができます。

`mmap` システム・コールによるファイルのメモリへのマッピングがシステム上において有効な場合、もう 1 つのオプション `-mapped` を使って、GDB に対して、再利用可能なファイルの中にユーザ・プログラムのシンボルを書き込ませることができます。後の GDB デバッグ・セッションは、(プログラムに変更がない場合) 実行プログラムからシンボル・テーブルを読み込むのに時間を費やすことなく、この補助シンボル・ファイルからシンボル情報をマップします。`-mapped` オプションを使用することは、コマンドライン・オプション `-mapped` を指定して GDB を起動するのと同じ効果を持ちます。



補助シンボル・ファイルがユーザ・プログラムのシンボル情報をすべて確実に持つように、両方のオプションを同時に指定することもできます。

*myprog* という名前のプログラムの補助シンボル・ファイルは、*‘myprog.syms’* という名前になります。このファイルが存在すると、(それが、対応する実行ファイルよりも新しい限り)ユーザが *myprog* をデバッグしようとする、GDB は常にそのファイルを使おうとします。特別なオプションやコマンドは必要ありません。

*‘.syms’* ファイルは、GDB を実行したホスト・マシンに固有のものです。それは、GDB 内部におけるシンボル・テーブルの正確なイメージを保持しています。複数のホスト・プラットフォーム間で共用することはできません。

`core-file [ filename ]`

「メモリ上のイメージ」として使用されるコア・ダンプ・ファイルの存在場所を指定します。伝統的に、コア・ファイルは、それを生成したプロセスのアドレス空間の一部だけを保持しています。GDB は、実行ファイルそのものにアクセスすることによって、保持されていない部分を獲得することができます。

`core-file` を引数なしで実行すると、コア・ファイルを一切使用しないことを指定したことになります。

ユーザ・プログラムが実際に GDB の管理下で実行中の場合は、コア・ファイルは無視されることに注意してください。したがって、ある時点までユーザ・プログラムを実行させた後に、コア・ファイルをデバッグしたくなったような場合、プログラムを実行しているサブ・プロセスを終了させなければなりません。サブ・プロセスの終了は、`kill` コマンドで行います (セクション 4.8 [子プロセスの終了], ページ 26 参照)。

`add-symbol-file filename address`

`add-symbol-file filename address [ -readnow ] [ -mapped ]`

`add-symbol-file filename address data_address bss_address`

`add-symbol-file filename -Tsection address`

`add-symbol-file` コマンドは、*filename* で指定されるファイルから追加的なシンボル・テーブル情報を読み込みます。*filename* で指定されるファイルが (何か別の方法によって) 実行中のプログラムの中に動的にロードされた場合に、このコマンドを使用します。*address* は、ファイルがロードされたメモリ・アドレスでなければなりません。GDB は独力でこのアドレスを知ることはできません。*address* は最大で 3 個まで指定することができます。3 個のアドレスを指定した場合、それらはそれぞれ、`text` セグメント、`data` セグメント、`bss` セグメントのアドレスとみなされます。より複雑な場合には、明示的にセクション名とそのセクションのベース・アドレスを指定するために、*‘-Tsection address’* を任意の数だけ指定することができます。*address* は式として指定することもできます。

*filename* で指定されるファイルのシンボル・テーブルは、もともと `symbol-file` コマンドによって読み込まれたシンボル・テーブルに追加されます。`add-symbol-file` コマンドは何回でも使用することができます。新たに読み込まれたシンボル・テーブルのデータは、古いデータに追加されていきます。古いシンボル・データをすべて破棄するには、引数を指定せずに `symbol-file` コマンドを使用してください。

`add-symbol-file` コマンドを実行した後に `(RET)` キーを押しても、`add-symbol-file` コマンドは繰り返し実行されません。

symbol-file コマンドと同様、'-mapped' オプションと '-readnow' オプションを使用して、*filename* で指定されるファイルのシンボル・テーブル情報を GDB がどのように管理するかを変更することができます。

#### add-shared-symbol-file

add-shared-symbol-file コマンドは、Motorola 88k 用の Harris' CXUX オペレーティング・システム上でのみ使用することができます。GDB は自動的に共有ライブラリを探しますが、GDB がユーザの共有ライブラリを見つけてくれない場合には、add-shared-symbol-file コマンドを実行できます。このコマンドは引数を取りません。

**section** section コマンドは、実行ファイルの *section* セクションのベース・アドレスを *addr* に変更します。これは、( a.out フォーマットのように ) 実行ファイルがセクション・アドレスを保持していない場合や、ファイルの中で指定されているアドレスが誤っている場合に使うことができます。個々のセクションは、個別に変更されなければなりません。以下において説明する info files コマンドによって、すべてのセクションとそのアドレスを一覧表示することができます。

#### info files

#### info target

info files と info target は同義です。両方とも、カレント・ターゲット ( 章 13 [デバッグ・ターゲットの指定], ページ 115 参照 ) に関する情報を表示します。表示される情報には、GDB が現在使用中の実行ファイルやコア・ダンプ・ファイルの名前、シンボルがそこからロードされたファイルの名前が含まれます。help target コマンドは、カレントなターゲットではなく、すべての可能なターゲットを一覧表示します。

ファイルを指定するすべてのコマンドは、引数として、絶対パスによるファイル名と相対パスによるファイル名のどちらでも受け付けます。GDB は、常にファイル名を絶対パス名に変換して、絶対パスの形で記憶します。

GDB は、HP-UX、SunOS、SVr4、Irix 5、IBM RS/6000 の共有ライブラリをサポートします。

ユーザが run コマンドを実行したり、コア・ファイルを調べようとすると、GDB は自動的に共有ライブラリからシンボル定義をロードします ( ユーザが run コマンドを発行するまでは、共有ライブラリ内部の関数への参照があっても、GDB にはそれを理解することができません。コア・ファイルをデバッグしている場合は、この限りではありません )。

HP-UX 上では、プログラムがライブラリを明示的にロードする場合は、shl\_load の呼び出しの時点において、GDB が自動的にシンボルをロードします。

#### info share

#### info sharedlibrary

現在ロードされている共有ライブラリの名前を表示します。

#### sharedlibrary *regex*

#### share *regex*

UNIX の正規表現にマッチするファイルに対応する、共有オブジェクト・ライブラリのシンボルをロードします。自動的にロードされるファイルと同様、ユーザ・プログラムによってコア・ファイルのために必要とされる共有ライブラリ、または run コマンド実行時に必要とされる共有ライブラリだけがロードされます。*regex* が省

略されると、ユーザ・プログラムによって必要とされるすべての共有ライブラリがロードされます。

HP-UX システムでは、GDB は共有ライブラリのローディングを検出し、新しくロードされたライブラリから自動的にシンボルを読み込みます。この読み込みは、ある上限まで行われます。この上限の値は、最初にセットされますが、そうしたければ変更することも可能です。

上限を超えた共有ライブラリのシンボルは、明示的にロードされなければなりません。これらのシンボルをロードするには、`sharedlibrary filename` コマンドを使用します。共有ライブラリのベース・アドレスは GDB によって自動的に決定されるので、指定する必要はありません。

上限を表示またはセットするには、以下のコマンドを使用します。

```
set auto-solib-add threshold
```

自動ロードするサイズの上限を、メガバイト単位でセットします。*threshold* の値がゼロ以外であれば、すべての共有オブジェクト・ライブラリのシンボルは、下記プロセスが実行を開始したとき、または、ダイナミック・リンクが、新しいライブラリがロードされたことを GDB に通知したときに、そのプログラムとライブラリのシンボル・テーブルのサイズがこの上限を超えるまで、自動的にロードされます。*threshold* の値がゼロであれば、シンボルは、`sharedlibrary` コマンドを使用して、手作業でロードしなければなりません。デフォルトの上限は、100 メガバイトです。

```
show auto-solib-add
```

現在の自動ロードのサイズの上限を、メガバイト単位で表示します。

## 12.2 シンボル・ファイル読み込み時のエラー

シンボル・ファイルの読み込み中に、GDB はときどき問題にぶつかることがあります。例えば、認識できないシンボル・タイプを見つけたり、コンパイラの出力に既知の問題を発見することがあります。デフォルトでは、このようなエラーがあったことを、GDB はユーザに知らせません。なぜなら、このようなエラーは比較的によく見られるものであり、コンパイラのデバッグをしているような人々だけが関心を持つようなものだからです。もし、正しく構築されていないシンボル・テーブルに関する情報を見ることに関心があれば、`set complaints` コマンドを使用することで、問題が何回発生しようと個々のタイプの問題について 1 回だけメッセージを出力するよう指示することができますし、また、問題が何回発生したかを見るためにより多くのメッセージを表示するよう指示することもできます (セクション 15.6 [オプションの警告およびメッセージ], ページ 160 参照)。

現在のバージョンで表示されるメッセージとその意味を以下に記します。

```
inner block not inside outer block in symbol
```

シンボル情報は、シンボルのスコープの先頭と末尾の位置を示します (例えば、ある関数の先頭、あるいは、ブロックの先頭など)。このエラーは、内側のスコープのブロックが、外側のスコープのブロックに完全に包含されていないことを意味しています。

GDB は、内側のブロックが外側のブロックと同一のスコープを持つものとして扱うことで、この問題を回避します。外側のブロックが関数でない場合には、エラー・メッセージの *symbol* の部分が ‘(don't know)’ のように表示されることがあります。

**block at *address* out of order**

シンボルのスコープとなるブロックに関する情報は、アドレスの低い方から昇順に並んでいなければなりません。このエラーは、そうになっていないことを示しています。

GDBはこの問題を回避することはせず、読み込もうとしているソース・ファイルのシンボルを見つけるのに支障が出ます ( `set verbose on` を指定することで、どのソース・ファイルが関係しているかを知ることができます。セクション 15.6 [オプションの警告およびメッセージ], ページ 160 を参照してください)。

**bad block start address patched**

シンボルのスコープとなるブロックに関する情報の中の開始アドレスが、1 つ前のソース行のアドレスより小さい値です。これは、SunOS 4.1.1 ( および、それ以前のバージョン ) の C コンパイラで発生することが分かっています。

GDB は、シンボルのスコープとなるブロックが 1 つ前のソース行から始まるものとして扱うことによって、この問題を回避します。

**bad string table offset in symbol *n***

シンボル番号 *n* のシンボルが持っている文字列テーブルへのポインタが、文字列テーブルのサイズを超える値です。

GDB は、このシンボルが `foo` という名前を持つものとみなすことによって、この問題を回避します。この結果、多くのシンボルが `foo` という名前を持つことになってしまうと、他の問題が発生する可能性があります。

**unknown symbol type *0xnn***

シンボル情報の中に、どのようにして読み取ればよいのか GDB には分からないような、新しいデータ型が含まれています。 `0xnn` は理解できなかったシンボルの型を 16 進数で表わしたものです。

GDB は、このようなシンボル情報を無視することによって、このエラーを回避します。通常、プログラムのデバッグを行うことは可能になりますが、特定のシンボルにアクセスすることができなくなります。このような問題にぶつかり、それをデバッグしたいのであれば、gdb 自身を使って gdb をデバッグすることができます。この場合、シンボル `complain` にブレイクポイントを設定し、関数 `read_dbx_syntab` まで実行してから、`*bufp` によってシンボルを参照します。

**stub type has NULL name**

GDB は、ある構造体またはクラスに関する完全な定義を見つけることができませんでした。

**const/volatile indicator missing (ok if using g++ v1.x), got...**

ある C++ のメンバ関数に関するシンボル情報に、より新しいコンパイラを使用した場合には生成されるいくつかの情報が欠けています。

**info mismatch between compiler and debugger**

GDB は、コンパイラが生成した型の指定を解析できませんでした。

## 13 デバッグ・ターゲットの指定

ターゲットとは、ユーザ・プログラムが持つ実行環境を指します。

多くの場合、GDB はユーザ・プログラムと同一のホスト環境上で実行されます。この場合には、`file` コマンドや `core` コマンドを実行すると、その副作用としてデバッグ・ターゲットが指定されます。例えば、物理的に離れた位置にあるホスト・マシン上で GDB を実行したい場合や、シリアル・ポート経由でスタンドアロン・システムを制御したい場合、または、TCP/IP 接続を利用してリアルタイム・システムを制御したい場合などのように、より多くの柔軟性が必要とされる場合、`target` コマンドを使うことによって、`configure` によって設定された GDB のターゲットの種類の中から 1 つを指定することができます (セクション 13.2 [ターゲットを管理するコマンド], ページ 115 参照)。

### 13.1 アクティブ・ターゲット

ターゲットには 3 つのクラスがあります。プロセス、コア・ファイル、そして、実行ファイルです。GDB は同時に、1 クラスにつき 1 つ、全体で最高で 3 つまでアクティブなターゲットを持つことができます。これにより、(例えば) コア・ファイルに対して行ったデバッグ作業を破棄することなく、プロセスを起動してその動作を調べることができます。

例えば、`'gdb a.out'` を実行すると、実行ファイル `a.out` が唯一のアクティブなターゲットになります。コア・ファイル (おそらくは、前回実行したときにクラッシュしてコア・ダンプしたもの) を併せて指定すると、GDB は 2 つのターゲットを持ち、メモリ・アドレスを知る必要がある場合には、それを知るために 2 つのターゲットを並行して使用します。この場合、まずコア・ファイルを参照し、次に実行ファイルを参照します。(典型的には、これら 2 つのクラスのターゲットは相互に補完的です。というのも、コア・ファイルには、プログラムが持っている変数などの読み書き可能なメモリ域の内容とマシン・ステータスだけがあり、実行ファイルには、プログラムのテキストと初期化されたデータだけがあるからです)。

`run` コマンドを実行すると、ユーザの実行ファイルはアクティブなプロセス・ターゲットにもなります。プロセス・ターゲットがアクティブな間は、メモリ・アドレスを要求するすべての GDB コマンドは、プロセス・ターゲットを参照します。アクティブなコア・ファイル・ターゲットや実行ファイル・ターゲットの中のアドレスは、プロセス・ターゲットがアクティブな間は、隠された状態になります。

新しいコア・ファイル・ターゲットや実行ファイル・ターゲットを選択するには、`core-file` コマンドや `exec-file` コマンドを使用します (セクション 12.1 [ファイルを指定するコマンド], ページ 109 参照)。既に実行中のプロセスをターゲットとして指定するには、`attach` コマンドを使用します (セクション 4.7 [既に実行中のプロセスのデバッグ], ページ 25 参照)。

### 13.2 ターゲットを管理するコマンド

`target type parameters`

GDB のホスト環境をターゲット・マシンまたはターゲット・プロセスに接続します。ターゲットとは、典型的には、デバッグ機能と通信するためのプロトコルを指します。引数 `type` によって、ターゲット・マシンの種類またはプロトコルを指定します。

`parameters` はターゲット・プロトコルによって解釈されるものですが、典型的には、接続すべきデバイス名やホスト名、プロセス番号、ボーレートなどが含まれます。

targetコマンドを実行した後に`(RET)`キーを押しても、targetコマンドは再実行されません。

#### help target

利用可能なすべてのターゲットの名前を表示します。現在選択されているターゲットを表示させるには、`info target`コマンドまたは`info files`コマンドを使用します（セクション 12.1 [ファイルを指定するコマンド], ページ 109 参照）。

#### help target *name*

ある特定のターゲットに関する説明を表示します。選択時に必要となるパラメータも表示されます。

#### set gnutarget *args*

GDB は、自分で持っているライブラリ BFD を使用してユーザ・ファイルを読み取ります。GDB は、実行ファイル、コア・ファイル、`.o` ファイルのどれを自分が読み取っているのかを知っています。しかし、`set gnutarget`コマンドを使用して、ファイルのフォーマットを指定することもできます。ほとんどの target コマンドとは異なり、`gnutarget`における target は、マシンではなくプログラムです。

注意：`set gnutarget`でファイル・フォーマットを指定するには、実際の BFD 名を知っている必要があります。

セクション 12.1 [ファイルを指定するコマンド], ページ 109 を参照してください。

#### show gnutarget

`gnutarget`がどのようなファイル・フォーマットを読むよう設定されているかを表示させるには、`show gnutarget`コマンドを使用します。`gnutarget`を設定していない場合、個々のファイルのフォーマットを GDB が自動的に決定します。この場合、`show gnutarget`を実行すると“The current BDF target is “auto””と表示されます。

以下に、一般的なターゲットをいくつか示します（GDB の構成によって、利用可能であったり利用不可であったりします）。

#### target exec *program*

実行ファイルです。‘target exec *program*’は‘exec-file *program*’と同じです。

#### target core *filename*

コア・ダンプ・ファイルです。‘target core *filename*’は‘core-file *filename*’と同じです。

#### target remote *dev*

GDB 固有のプロトコルによる、リモートのシリアル・ターゲットです。引数 *dev* によって、接続を確立するために使用するシリアル装置（例えば、‘/dev/ttya’）を指定します。セクション 13.4 [リモート・デバッグ], ページ 118 を参照してください。`target remote`は、`load`コマンドをサポートしています。これは、スタブをターゲット・システム上に持っていく方法が別にあり、かつ、ダウンロードが実行されたときに破壊されないようなメモリ域にそれを置くことができる場合のみ役に立ちます。

#### target sim

組み込み CPU シミュレータです。GDB には、ほとんどのアーキテクチャ用のシミュレータが含まれています。一般には、

```
target sim
load
run
```

うまくいきます。しかし、シミュレータによっては特定のメモリ・マップ、デバイス・ドライバ、基本的な I/O を提供するものもありますが、それらが利用可能であることを前提することはできません。プロセッサ固有のシミュレータの詳細に関しては、セクション 14.3 [Embedded Processors], ページ 142 の該当するセクションを参照してください。

構成によっては、以下のターゲットも含まれている可能性があります。

```
target nrom dev
```

NetROM ROM エミュレータです。このターゲットは、ダウンロードのみサポートしています。

configure による GDB の構成によって、利用可能なターゲットも異なるものになります。configure の構成次第で、ターゲットの数は多くなったり少なくなったりします。

多くのリモート・ターゲットでは、接続に成功すると、実行プログラムのコードをダウンロードすることが必要となります。

```
load filename
```

configure の実行時に GDB に組み込まれたリモート・デバッグ機能によっては、load コマンドが使用可能になります。これが利用可能な場合、実行ファイル *filename* が (例えば、ダウンロードやダイナミック・リンクによって) リモート・システム上でデバッグできるようになることを意味します。また、load コマンドは add-symbol-file コマンドと同様、ファイル *filename* のシンボル・テーブルを GDB 内に記録します。

GDB が load コマンドを提供していない場合、それを実行しようとすると「You can't do that when your target is ...」というエラー・メッセージが表示されます。

実行ファイルの中で指定されたアドレスに、ファイルはロードされます。オブジェクト・ファイルのフォーマットによっては、プログラムをリンクするときに、ファイルをロードするアドレスを指定できるものもあります。これ以外のフォーマット (例えば、a.out) では、オブジェクト・ファイルのフォーマットによって固定的にアドレスが指定されます。

load コマンドを実行後に **RET** キーを押しても、コマンドは再実行されません。

### 13.3 ターゲットのバイト・オーダの選択

MIPS、PowerPC、Hitachi SH などのプロセッサは、ビッグ・エンディアン、リトル・エンディアンのどちらのバイト・オーダでも実行することができます。通常は、実行ファイルまたはシンボルの中に、エンディアン種別を指定するビットがあるので、どちらを使用するかを気にする必要はありません。しかし、GDB の認識しているプロセッサのエンディアン種別を手作業で調整することができれば、便利なこともあるでしょう。

```
set endian big
```

GDB に対して、ターゲットはビッグ・エンディアンであると想定するよう指示します。

```
set endian little
```

GDB に対して、ターゲットはリトル・エンディアンであると想定するよう指示します。

```
set endian auto
```

GDB に対して、実行ファイルに関連付けされているバイト・オーダーを使用するよう指示します。

```
show endian
```

GDB が認識している、ターゲットの現在のバイト・オーダー種別を表示します。

これらのコマンドは、ホスト上でのシンボリック・データの解釈を調整するだけであり、ターゲット・システムに対しては全く何の影響も持たないということに注意してください。

### 13.4 リモート・デバッグ

通常の方法で GDB を実行することのできないマシン上で実行されているプログラムをデバッグするには、リモート・デバッグ機能を使うのが便利です。例えば、オペレーティング・システムのカーネルのデバッグや、フル機能を持つデバッガを実行するのに十分な機能を持つ汎用的なオペレーティング・システムを持たない小規模なシステムでのデバッグでは、ユーザはリモート・デバッグ機能を使うことになるかもしれません。

GDB は、その構成によっては、特別なシリアル・インターフェイスや TCP/IP インターフェイスを持ち、これを特定のデバッグ・ターゲット用に使用することができます。さらに、GDB には汎用的なシリアル・プロトコルが組み込まれており (GDB 固有のもので、特定のターゲット・システムに固有なものではありません)、リモート・スタブを作成すれば、これを使用することができます。リモート・スタブとは、GDB と通信するためにリモート・システム上で動作するコードです。

GDB の構成によっては、他のリモート・ターゲットが利用可能な場合もあります。利用可能なリモート・ターゲットを一覧表示させるには、`help target` コマンドを使用します。

#### 13.4.1 GDB リモート・シリアル・プロトコル

他のマシン上で実行中のプログラムをデバッグするには (ターゲット・マシンをデバッグするには)、そのプログラムを単独で実行するために通常必要となる事前条件をすべて整える必要があります。例えば、C のプログラムの場合、

1. C の実行環境をセットアップするためのスタートアップ・ルーチンが必要です。これは通常 `'crt0'` のような名前を持っています。スタートアップ・ルーチンは、ハードウェアの供給元から提供されることもありますし、ユーザが自分で書かなければならないこともあります。
2. ユーザ・プログラムからのサブルーチン呼び出しをサポートするために、入出力の管理などを行う C のサブルーチン・ライブラリが必要になるかもしれません。
3. ユーザ・プログラムを他のマシンに持っていく手段、例えばダウンロード・プログラムが必要です。これはハードウェアの供給元から提供されることが多いのですが、ハードウェアのドキュメントをもとにユーザが自分で作成しなければならないこともあります。

次に、ユーザ・プログラムがシリアル・ポートを使って、GDB を実行中のマシン (ホスト・マシン) と通信できるように準備します。一般的には、以下のような形になります。



ホスト上では：

GDB は既にこのプロトコルの使い方を理解しています。他の設定がすべて終了した後、単に ‘target remote’ コマンドを使用するだけです ( 章 13 [デバッグ・ターゲットの指定], ページ 115 参照 )。

ターゲット上では：

ユーザ・プログラムに、GDB リモート・シリアル・プロトコルを実装した特別なサブルーチンをいくつかリンクする必要があります。これらのサブルーチンを含むファイルは、デバッグ・スタブと呼ばれます。

特定のリモート・ターゲットでは、ユーザ・プログラムにスタブをリンクする代わりに、gdbserver という補助プログラムを使うこともできます。詳細については、セクション 13.4.1.5 [gdbserver プログラムの使用], ページ 134 を参照してください。

デバッグ・スタブはリモート・マシンのアーキテクチャに固有のものです。例えば、SPARC ボード上のプログラムをデバッグするには ‘sparc-stub.c’ を使います。

以下に実際に使えるスタブを列挙します。これらは、GDB とともに配布されています。

i386-stub.c

Intel 386 アーキテクチャ、およびその互換アーキテクチャ用です。

m68k-stub.c

Motorola 680x0 アーキテクチャ用です。

sh-stub.c

日立 SH アーキテクチャ用です。

sparc-stub.c

SPARC アーキテクチャ用です。

sparcl-stub.c

富士通 SPARCLITE アーキテクチャ用です。

GDB とともに配布される README ファイルには、新しく追加された他のスタブのことが記されているかもしれません。

#### 13.4.1.1 スタブの提供する機能

各アーキテクチャ用のデバッグ・スタブは、3 つのサブルーチンを提供します。

set\_debug\_traps

このルーチンは、ユーザ・プログラムが停止したときに handle\_exception が実行されるよう設定します。ユーザ・プログラムは、その先頭付近でこのサブルーチンを明示的に呼び出さなければなりません。

handle\_exception

これが中心的な仕事をする部分ですが、ユーザ・プログラムはこれを明示的には呼び出しません。セットアップ・コードによって、トラップが発生したときに handle\_exception が実行されるよう設定されます。

ユーザ・プログラムが実行中に ( 例えば、ブレイクポイントで ) 停止すると、handle\_exception が制御権を獲得し、ホスト・マシン上の GDB との通信を行います。これが、通信プロトコルが実装されている部分です。handle\_exception は、ター

ゲット・マシン上で GDB の代理として機能します。それはまず、ユーザ・プログラムの状態に関する情報を要約して送ることから始めます。次に、GDB が必要とする情報を入手して転送する処理を継続します。これは、ユーザ・プログラムの実行を再開させるような GDB コマンドが実行されるまで続きます。そのようなコマンドが実行されると、`handle_exception`は、制御をターゲット・マシン上のユーザ・コードに戻します。

#### breakpoint

ユーザ・プログラムにブレイクポイントを持たせるには、この補助的なサブルーチンを使います。特定の状況においては、これが GDB が制御を獲得する唯一の方法です。例えば、ユーザのターゲット・マシンに割り込みを発生させるボタンのようなものがあれば、このサブルーチン呼び出す必要はありません。割り込みボタンを押すことで、制御は `handle_exception`に、つまり事実上 GDB に渡されます。マシンによっては、シリアル・ポートから文字を受け取るだけでトラップが発生することもあります。このような場合には、ユーザ・プログラム自身から breakpoint を呼び出す必要はなく、ホストの GDB セッションから `'target remote'`を実行するだけで制御を得ることができます。

これらのどのケースにも該当しない場合、あるいは、デバッグ・セッションの開始箇所としてあらかじめ決めてあるところでユーザ・プログラムが停止することを単に確実にしたいのであれば、`breakpoint`を呼び出してください。

### 13.4.1.2 スタブに対する必須作業

GDB とともに配布されるデバッグ用スタブは、特定のチップのアーキテクチャ用にセットアップされたものですが、デバッグのターゲット・マシンに関してそれ以外の情報は持っていません。まず最初に、どのようにしてシリアル・ポートと通信するかをスタブに教えてやる必要があります。

#### `int getDebugChar()`

シリアル・ポートから単一文字を読み取るサブルーチンとしてこれを書きます。これは、ターゲット・システム上の `getchar`と同一かもしれませんが。これら 2 つを区別したい場合を考慮して、異なる名前が使われています。

#### `void putDebugChar(int)`

シリアル・ポートに単一文字を書き込むサブルーチンとしてこれを書きます。これは、ターゲット・システム上の `putchar`と同一かもしれませんが。これら 2 つを区別したい場合を考慮して、異なる名前が使われています。

実行中のユーザ・プログラムを GDB が停止できるようにしたいのであれば、割り込み駆動型のシリアル・ドライバを使用して、`^C` ( `control-C` 文字、すなわち `'\003'` )を受信したときに停止するよう設定する必要があります。GDB は`^C`を使って、リモート・システムに対して停止するよう通知します。

デバッグ・ターゲットが適切なステータス情報を GDB に対して返せるようにするためには、おそらく標準のスタブを変更する必要があるでしょう。美しい方法ではありませんが、とりあえず手っ取り早くこれを実現する方法は、ブレイクポイント命令を実行することです(この方法が「美しい」のは、GDB が `SIGINT`ではなく `SIGTRAP`を報告してくる点にあります)。

ユーザが提供する必要のあるルーチンには、ほかに以下のようなものがあります。

```
void exceptionHandler (int exception_number, void *exception_address)
```

例外処理テーブルに *exception\_address* を組み込むよう、この関数を書きます。ユーザがこれを提供しなければならないのは、スタブにはターゲット・システム上の例外処理テーブルがどのようなものになるかを知る手段がないからです（例えば、プロセッサのテーブルは ROM 上にあり、その中のエントリが RAM 上のテーブルを指す、という形になっているかもしれません）。*exception\_number* は例外番号で、これは変更される必要があります。例外番号の意味は、アーキテクチャに依存します（例えば、0 による除算、境界を無視したメモリ・アクセス等は、異なる番号によって表わされるかもしれません）。この例外が発生したとき、制御は直接 *exception\_address* に渡されなければならず、また、プロセッサの状態（スタック、レジスタなど）はプロセッサ例外が発生したときの状態と同じでなければなりません。したがって、*exception\_address* に到達するのにジャンプ命令を使用したいのであれば、サブルーチン・ジャンプではなく、ただのジャンプ命令を使わなければなりません。

386 では、ハンドラが実行されているときに割り込みがマスクされるよう、*exception\_address* は割り込みゲートとして組み込まれる必要があります。そのゲートは特権レベル 0（最も高いレベル）でなければなりません。SPARC 用のスタブや 68k 用のスタブは、*exceptionHandler* の助けを借りなくても自分で割り込みをマスクすることができます。

```
void flush_i_cache()
```

（SPARC、SPARCLITE のみ）ターゲット・マシンに命令キャッシュがある場合、それをフラッシュするようこのサブルーチンを書きます。命令キャッシュがない場合には、このサブルーチンは何もしないものになるかもしれません。

命令キャッシュを持つターゲット・マシンを対象としている場合、GDB は、ユーザ・プログラムが安定した状態にあることをこの関数が保証してくれることを必要とします。

また、次のライブラリ・ルーチンが使用可能であることを確かめなければなりません。

```
void *memset(void *, int, int)
```

あるメモリ領域に既知の値を設定する標準ライブラリ関数 *memset* です。フリーの *libc.a* を持っていれば、そこに *memset* があります。フリーの *libc.a* がなければ、*memset* をハードウェアの供給元から入手するか、自分で作成する必要があります。

GNU C コンパイラを使っていないのであれば、他の標準ライブラリ・サブルーチンも必要になるかもしれません。これは、スタブによっても異なりますが、一般的にスタブは、*gcc* がインライン・コードとして生成する共通ライブラリ・サブルーチンを使用する可能性があります。

### 13.4.1.3 ここまでのまとめ

要約すると、ユーザ・プログラムをデバッグする準備が整った後、以下の手順に従わなければなりません。

1. 下位レベルのサポート・ルーチンがあることを確認します（セクション 13.4.1.2 [スタブに対する必須作業]、ページ 120 参照）。

```
getDebugChar, putDebugChar,  
flush_i_cache, memset, exceptionHandler.
```

2. ユーザ・プログラムの先頭付近に以下の行を挿入します。

```
set_debug_traps();
breakpoint();
```

3. 680x0 のスタブに限り、exceptionHookという変数を提供する必要があります。通常は、以下のように使います。

```
void (*exceptionHook)() = 0;
```

しかし、set\_debug\_trapsが呼び出される前に、ユーザ・プログラム内のある関数を指すようこの変数を設定すると、トラップ（例えば、バス・エラー）で停止した後に GDB が処理を継続実行するときに、その関数が呼び出されます。exceptionHookによって指される関数は、1 つの引数付きで呼び出されます。それは、int型の例外番号です。

4. ユーザ・プログラム、ターゲット・アーキテクチャ用の GDB デバッグ・スタブ、サポート・サブルーチンをコンパイルしリンクします。
5. ターゲット・マシンと GDB ホストとの間がシリアル接続されていることを確認します。また、ホスト上のシリアル・ポートの名前を調べます。
6. ターゲット・マシンにユーザ・プログラムをダウンロードし（あるいは、製造元の提供する手段によってターゲット・マシンにユーザ・プログラムを持っていき）、起動します。
7. リモート・デバッグを開始するには、ホスト・マシン上で GDB を実行し、リモート・マシン上で実行中のプログラムを実行ファイルとして指定します。これにより、ユーザ・プログラムのシンボルとテキスト域の内容を見つける方法が GDB に通知されます。
8. 次に target remoteコマンドを使って通信を確立します。引数には、シリアル回線に接続された装置名または（通常はターゲットと接続されたシリアル回線を持つ端末サーバの）TCP ポートを指定することで、ターゲット・マシンとの通信方法を指定します。例えば、`‘/dev/ttyb’`という名前の装置に接続されているシリアル回線を使うには、

```
target remote /dev/ttyb
```

とします。

TCP 接続を使うには、`host:port`という形式の引数を使用します。例えば、manyfarmsという名前の端末サーバのポート 2828 に接続するには、

```
target remote manyfarms:2828
```

とします。

ここまでくると、データの値の調査、変更、リモート・プログラムのステップ実行、継続実行に通常使用するすべてのコマンドを使用することができます。

リモート・プログラムの実行を再開し、デバッグするのをやめるには、detachコマンドを使います。

GDB がリモート・プログラムを待っているときにはいつでも、割り込み文字（多くの場合 `C-C`）を入力すると、GDB はそのプログラムを停止しようとします。これは成功することも失敗することもあります。その成否は、リモート・システムのハードウェアやシリアル・ドライバにも依存します。割り込み文字を再度入力すると、GDB は以下のプロンプトを表示します。

```
Interrupted while waiting for the program.
Give up (and stop debugging it)? (y or n)
```

ここで `y`を入力すると、GDB はリモート・デバッグ・セッションを破棄します（後になって再実行したくなった場合には、接続するために `‘target remote’`を再度使用します）。`n`を入力すると、GDB は再び待ち状態になります。

#### 13.4.1.4 通信プロトコル

GDB とともに提供されるスタブ・ファイルは、ターゲット側の通信プロトコルを実装します。そして GDB 側の通信プロトコルは、GDB のソース・ファイル `'remote.c'` に実装されています。通常は、これらのサブルーチンに通信処理を任せて、詳細を無視することができます（独自のスタブ・ファイルを作成するときでも、詳細については無視して、既存のスタブ・ファイルをもとにして作成を始めることができます。`'sparc-stub.c'` が最もよく整理されており、したがって最も読みやすくなっています）。

しかし、場合によっては、プロトコルについて何かを知る必要が出てくることもあるでしょう。例えば、ターゲット・マシンにシリアル・ポートが 1 つしかなく、GDB に対して送られてきたパケットを検出したときに、ユーザ・プログラムが何か特別なことをするようにしたい場合です。以下の例では、`'<-'` と `'->'` が、それぞれ転送されたデータと受信されたデータを示すために使われています。

（確認メッセージを除く）すべての GDB コマンドとそれに対する応答は、`packet`（パケット）として送信されます。`packet` は、文字 `'$'` で始まり、その後ろに実際の `packet-data`（パケット・データ）が続きます。さらにその後ろに、終端文字 `'#'` と 2 桁のチェックサム値が続きます。

```
$packet-data#checksum
```

ここで、2 桁の数字から構成される `checksum` は、先頭の `'$'` と終端の `'#'` の間にあるすべての文字の値を合計したものを 256 で割った余りとして計算されます（8 ビットの無符号チェックサム）。プロトコル実装者は、GDB 5.0 よりも古いバージョンのプロトコル仕様においては、必須ではないものの、2 桁の数字の `sequence-id`（シーケンス ID）が含まれていたことに注意する必要があります。

```
$sequence-id:packet-data#checksum
```

この `sequence-id` は確認メッセージの末尾に付加されていました。GDB が `sequence-id` を出力するようになっていたことは過去に一度もありません。GDB 5.0 以降に追加されたパケットを処理するスタブは、`sequence-id` を受け付けてはなりません。

ホスト・マシンまたはターゲット・マシンがパケットを受信したとき、最初に期待される応答は確認メッセージです。これは単一文字で、（パッケージが正しく受信されたことを示す）`'+'` または（再送要求を示す）`'-'` です。

```
<- $packet-data#checksum
-> +
```

ホスト（GDB）が `command`（コマンド）を送信し、ターゲット（ユーザ・プログラムに組み込まれたデバッグ・スタブ）が `response`（応答）を送信します。ステップ実行コマンドや継続実行コマンドの場合には、操作が完了した（ターゲットが再び停止した）ときにのみ応答が送信されます。

`packet-data` は、`'#'` と `'$'` を除く任意の文字が連続したものです（`'X'` パケットについてはこれら以外にも除外される文字があります）。

パケット内部のフィールドは、`'.'`、`','`、`':'` のいずれかによって区切られなければなりません。特にことわりがない限り、数字はすべて 16 進数で表現し、先頭のゼロは記しません。

プロトコル実装者は、GDB 5.0 よりも古いバージョンでは、（`sequence-id` と潜在的に矛盾するため）パケット内部の 3 番目の文字が文字 `':'` であってはならなかったことに注意するべきです。

領域を節約するために、ある文字が繰り返し現われる回数を符号化する方法を使って応答データ（`data`）を符号化することができます。`'*'` は、その次の文字が、`'*'` の前にある文字の繰り返し回

数を ASCII 符号化したものであることを意味しています。繰り返し回数の符号化の結果は  $n+29$  となります。29を加算することにより、(繰り返し回数  $n$  を符号化する方法が有意義である)  $n \geq 3$  の場合において、 $n$  に対応する表示可能文字が生成されます。表示可能文字 '\$'、'#'、'+'、'-'、および、126 を超える数値は使ってはなりません。

いくつかのリモート・システムでは、繰り返し回数を符号化する別のメカニズムが使われていました。これは、大雑把にシスコ符号化と呼ばれています。この場合は、文字 '\*' の後ろには、パケットのサイズを示す 2 桁の数字が続きます。

したがって、

```
"0* "
```

は、"0000" と同等のことを意味します。

いくつかのパケットに対して返されるエラー応答には 2 文字のエラー番号が含まれます。この番号は明確には定義されていません。

スタブがサポートしていないコマンド (*command*) に対しては、空の応答 ('\$#00') が返されるべきです。こうすれば、プロトコルの拡張が可能です。最新の GDB は、この応答を頼りにして、あるパケットがサポートされているかどうかを判断することができます。

スタブに対しては、'g'、'G'、'm'、'M'、'c'、's' の各コマンド (*command*) をサポートすることが要求されています。これら以外のコマンドはすべて必須ではありません。

以下に、現在定義されているすべてのコマンド (*command*) と対応する応答データ (*data*) の一覧を示します。

| パケット               | 要求                          | 説明                                                                                                                                                    |
|--------------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| 拡張 ops             | !                           | 拡張リモート・プロトコルを使用します。これは一度だけセットすれば十分であり、設定は持続します。拡張リモート・プロトコルは 'R' パケットをサポートしています。                                                                      |
|                    | 応答 ' '                      | 拡張リモート・プロトコルをサポートしているスタブは ' ' を返します。これは残念なことに、プロトコル拡張をサポートしていないスタブが返す応答と同一です。                                                                         |
| 最後のシグナル            | ?                           | ターゲットが停止した理由を示します。応答は、ステップ実行や継続実行の場合と同じです。                                                                                                            |
|                    | 応答                          | 後述                                                                                                                                                    |
| 予約済                | a                           | 将来のために予約されています。                                                                                                                                       |
| プログラム引数のセット (予約済)  | Aarglen, argnum, arg, . . . | プログラムに渡された 'argv []' 配列を初期化します。arglen には、16 進で符号化されたバイト・ストリームである arg のバイト数を指定します。詳細については 'gdbserver' を参照してください。                                       |
|                    | 応答 OK                       |                                                                                                                                                       |
|                    | 応答 ENN                      |                                                                                                                                                       |
| ボーレートのセット (非推奨)    | bbaud                       | シリアル回線のスピードを baud に変更します。                                                                                                                             |
| ブレイクポイントのセット (非推奨) | Baddr, mode                 | ( mode が 'S' の場合は ) addr によって指定されるアドレスにブレイクポイントをセットします。また、( mode が 'C' の場合は ) addr によって指定されるアドレスのブレイクポイントを削除します。これは、'Z' パケットと 'z' パケットによって取って代わられました。 |
| 継続実行               | caddr                       | addr は実行を再開すべきアドレスです。addr が省略された場合、現在のアドレスにおいて実行を再開します。                                                                                               |

|                | 応答               | 後述                                                                                                                                                                                                         |
|----------------|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| シグナルによる継続実行    | <i>Csig;addr</i> | ( 16 進数 ) <i>sig</i> によって示されるシグナル通知によって継続実行します。<br>; <i>addr</i> の部分が省略されると、現在のアドレスから実行を再開します。                                                                                                             |
|                | 応答               | 後述                                                                                                                                                                                                         |
| デバッグのトグル (非推奨) | d                | デバッグ・フラグをトグルします。                                                                                                                                                                                           |
| ディタッチ          | D                | リモート・システムから GDB をディタッチします。GDB が接続を切断する前に、リモート・ターゲットに送られます。                                                                                                                                                 |
|                | 応答なし             | このパケットを送った後、GDB はいかなる応答もチェックしません。                                                                                                                                                                          |
| 予約済            | e                | 将来のために予約されています。                                                                                                                                                                                            |
| 予約済            | E                | 将来のために予約されています。                                                                                                                                                                                            |
| 予約済            | f                | 将来のために予約されています。                                                                                                                                                                                            |
| 予約済            | F                | 将来のために予約されています。                                                                                                                                                                                            |
| レジスタの読み取り      | g                | 汎用レジスタの内容を読み取ります。                                                                                                                                                                                          |
|                | 応答 XX...         | レジスタ・データの個々のバイトは 2 桁の 16 進数字によって表現されます。レジスタのバイト情報はターゲットのバイト・オーダで転送されます。個々のレジスタのサイズと 'g' packet 中における位置は、GDB の内部マクロ <i>REGISTER_RAW_SIZE</i> および <i>REGISTER_NAME</i> によって決定されます。いくつかの標準 g パケットの仕様を以下に示します。 |
|                | ENN              | エラーを示します。                                                                                                                                                                                                  |
| レジスタへの書き込み     | GXX...           | XX... データの説明については 'g' の項を参照してください。                                                                                                                                                                         |
|                | 応答 OK            | 成功を示します。                                                                                                                                                                                                   |
|                | 応答 ENN           | エラーを示します。                                                                                                                                                                                                  |
| 予約済            | h                | 将来のために予約されています。                                                                                                                                                                                            |
| スレッドのセット       | Hc t...          | 後続のオペレーション ( 'm'、'M'、'g'、'G'、その他 ) のためのスレッドをセットします。                                                                                                                                                        |



|                                  |                             |                                                                                                                                        |
|----------------------------------|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
|                                  | 応答 OK                       | 成功を示します。                                                                                                                               |
|                                  | 応答 ENN                      | エラーを示します。                                                                                                                              |
| クロック・サイクルによるステップ実行 (ドラフト)        | <i>i addr, nnn</i>          | 単一クロック・サイクルによってリモート・ターゲットをステップ実行します。、 <i>nnn</i> の部分が指定されていれば、 <i>nnn</i> サイクルだけステップ実行します。 <i>addr</i> が指定されていれば、そのアドレスからステップ実行を開始します。 |
| シグナル通知後、クロック・サイクルによるステップ実行 (予約済) | I                           | 構文とその意味は、おそらく 'i' や 'S' と似たものになると思われます。これらの項を参照してください。                                                                                 |
| 予約済                              | j                           | 将来のために予約されています。                                                                                                                        |
| 予約済                              | J                           | 将来のために予約されています。                                                                                                                        |
| キル (kill) 要求                     | k                           |                                                                                                                                        |
| 予約済                              | l                           | 将来のために予約されています。                                                                                                                        |
| 予約済                              | L                           | 将来のために予約されています。                                                                                                                        |
| メモリの読み取り                         | <i>maddr, length</i>        | アドレス <i>addr</i> から始まる <i>length</i> バイトのメモリを読み取ります。GDB もスタブも、このメモリ転送がワード境界に境界整列されたアクセスを使うものと前提してはいません。                                |
|                                  | 応答 XX...                    | XX... はメモリの内容です。データの一部しか読むことができない場合は、要求されたバイト数よりも少ない可能性があります。GDB もスタブも、このメモリ転送がワード境界に境界整列されたアクセスを使うものと前提してはいません。                       |
|                                  | 応答 ENN                      | NN は <code>errno</code> です。                                                                                                            |
| メモリへの書き込み                        | <i>Maddr, length: XX...</i> | アドレス <i>addr</i> から始まるメモリに <i>length</i> バイトのデータを書き込みます。XX... はデータです。                                                                  |
|                                  | 応答 OK                       | 成功を示します。                                                                                                                               |
|                                  | 応答 ENN                      | エラーを示します (データの一部しか書き込めなかった場合も含まれます)。                                                                                                   |
| 予約済                              | n                           | 将来のために予約されています。                                                                                                                        |

|                 |                              |                                                                                                                                                                                                                                      |
|-----------------|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 予約済             | N                            | 将来のために予約されています。                                                                                                                                                                                                                      |
| 予約済             | o                            | 将来のために予約されています。                                                                                                                                                                                                                      |
| 予約済             | 0                            | 将来のために予約されています。                                                                                                                                                                                                                      |
| レジスタの読み取り (予約済) | p <i>n</i> ...               | 「レジスタへの書き込み」の項を参照してください。                                                                                                                                                                                                             |
|                 | 応答 <i>r</i> ...              | レジスタの値をターゲットのバイト・オーダで 16 進符号化したもの。                                                                                                                                                                                                   |
| レジスタへの書き込み      | P <i>n</i> ...= <i>r</i> ... | レジスタ <i>n</i> ... に値 <i>r</i> ... を書き込みます。<br><i>r</i> ... には、レジスタ内の個々のバイトについて 2 桁の 16 進数字が含まれます (ターゲットのバイト・オーダによる)。                                                                                                                 |
|                 | 応答 OK                        | 成功を示します。                                                                                                                                                                                                                             |
|                 | 応答 ENN                       | エラーを示します。                                                                                                                                                                                                                            |
| 一般的なクエリー        | q <i>query</i>               | クエリー (問い合わせ) <i>query</i> に関する情報を要求します。通常 GDB のクエリーは大文字で始まります。ベンダがカスタマイズしたクエリーには、例えば 'qfsf.var' のように、その企業を表わす (小文字の) 接頭語を使うべきです。 <i>query</i> の後ろには、',' または ';' によって区切られたリストが続くこともあります。スタブは、完全な <i>query</i> 名にマッチすることを保証しなければなりません。 |
|                 | 応答 XX...                     | クエリーの結果を 16 進符号化したデータです。応答は空であってはなりません。                                                                                                                                                                                              |
|                 | 応答 ENN                       | エラー応答です。                                                                                                                                                                                                                             |
|                 | 応答 '                         | <i>query</i> を理解できなかったことを示します。                                                                                                                                                                                                       |
| 一般的なセット         | Q <i>var</i> = <i>val</i>    | <i>var</i> の値を <i>val</i> にセットします。命名規則に関する議論については 'q' の項を参照してください。                                                                                                                                                                   |
| リセット (非推奨)      | r                            | システム全体をリセットします。                                                                                                                                                                                                                      |
| リモート再起動         | RXX                          | リモート・サーバを再起動します。XX の部分は必要ではありませんが、明確には定義されていません。                                                                                                                                                                                     |

|                             |                           |                                                                                                                                                 |
|-----------------------------|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| ステップ実行                      | <i>saddr</i>              | <i>addr</i> は実行を再開するアドレスです。<br><i>addr</i> が省略されると、現在のアドレスから実行を再開します。                                                                           |
|                             | 応答                        | 後述                                                                                                                                              |
| シグナルによるステップ実行               | <i>Ssig;addr</i>          | ‘C’と似ていますが、継続実行するのではなくステップ実行します。                                                                                                                |
|                             | 応答                        | 後述                                                                                                                                              |
| 検索                          | <i>taddr:PP,MM</i>        | アドレス <i>addr</i> からはじめて、パターン <i>PP</i> およびマスク <i>MM</i> にマッチするものを後方検索します。 <i>PP</i> と <i>MM</i> は 4 バイトです。 <i>addr</i> は少なくとも 3 桁の数字でなければなりません。 |
| スレッドの状態確認                   | <i>TXX</i>                | スレッド <i>XX</i> が存在する ( alive ) かどうかを調べます。                                                                                                       |
|                             | 応答 OK                     | スレッドはまだ存在します ( alive )                                                                                                                          |
|                             | 応答 ENN                    | スレッドはもう存在しません ( dead )                                                                                                                          |
| 予約済                         | <i>u</i>                  | 将来のために予約されています。                                                                                                                                 |
| 予約済                         | <i>U</i>                  | 将来のために予約されています。                                                                                                                                 |
| 予約済                         | <i>v</i>                  | 将来のために予約されています。                                                                                                                                 |
| 予約済                         | <i>V</i>                  | 将来のために予約されています。                                                                                                                                 |
| 予約済                         | <i>w</i>                  | 将来のために予約されています。                                                                                                                                 |
| 予約済                         | <i>W</i>                  | 将来のために予約されています。                                                                                                                                 |
| 予約済                         | <i>x</i>                  | 将来のために予約されています。                                                                                                                                 |
| メモリへの書き込み ( バイナリ )          | <i>Xaddr,length:XX...</i> | <i>addr</i> はアドレス、 <i>length</i> はバイト数、 <i>XX...</i> はバイナリ・データです。文字 \$、#、および 0x7d は 0x7d を使ってエスケープします。                                          |
|                             | 応答 OK                     | 成功を示します。                                                                                                                                        |
|                             | 応答 ENN                    | エラーを示します。                                                                                                                                       |
| 予約済                         | <i>y</i>                  | 将来のために予約されています。                                                                                                                                 |
| 予約済                         | <i>Y</i>                  | 将来のために予約されています。                                                                                                                                 |
| ブレイクポイント、ウォッチポイントの削除 (ドラフト) | <i>zt,addr,length</i>     | ‘Z’の項を参照してください。                                                                                                                                 |

ブレイクポイント、ウォッチポイントの挿入 (ドラフト)  $Zt, addr, length$

$t$  は以下の種類を示します。‘0’ - ソフトウェア・ブレイクポイント、‘1’ - ハードウェア・ブレイクポイント、‘2’ - ウォッチポイントへの書き込み、‘3’ - ウォッチポイントの読み取り、‘4’ - ウォッチポイントへのアクセス。 $addr$  はアドレスです。 $length$  はバイト数による長さです。ソフトウェア・ブレイクポイントの場合、 $length$  はパッチを当てられる命令のサイズを指定します。ハードウェア・ブレイクポイントおよびハードウェア・ウォッチポイントの場合、 $length$  は監視されるメモリ域を指定します。重複するバケットによる潜在的な問題を回避するために、オペレーションは副作用のないよう (idempotent) に実装しなければなりません。

応答 ENN

エラーを示します

応答 OK

成功を示します。

“

サポートされていない場合の応答です。

予約済

<other>

将来のために予約されています。

パケット ‘C’、‘c’、‘S’、‘s’、‘?’ は、以下のいずれであっても応答として受け取ることができません。パケット ‘C’、‘c’、‘S’、‘s’ の場合、この応答はターゲットが停止したときにはじめて返されます。以下において、‘signal number’ (シグナル番号) の正確な意味はきちんと定義されていません。一般的には、UNIX におけるシグナルの番号付けの慣習の 1 つが使われます。

SAA

AA はシグナル番号です。

TAAn...:r...;n...:r...;n...:r...;

AA は 2 桁の数字で表わされたシグナル番号です。 $n...$  は 16 進数のレジスタ番号です。 $r...$  はレジスタの内容をターゲットのバイト・オーダで表現したもので、そのサイズは REGISTER\_RAW\_SIZE によって定義されます。 $n...$  は ‘thread’ です。 $r...$  はそのスレッドのプロセス ID で、16 進整数です。 $n...$  は先頭の文字が正当な 16 進数字以外である文字列です。GDB は、この  $n...$  と  $r...$  のペアを無視して、次に進むはずですが、これにより、プロトコルの拡張が可能になります。

WAA

プロセスが終了し、その終了ステータスが AA です。これは、特定の種類のターゲットにのみ適用可能です。

XAA

プロセスはシグナル AA によって停止しました。

0AA;t...;d...;b... (旧)

AA はシグナル番号です。t... はシンボル "\_start" のアドレスです。d... は data セクションのベース・アドレスです。b... は bss セクションのベース・アドレスです。注: *Cisco Systems* 社のターゲットによってのみ利用されます。この応答と "qOffsets" クエリーの違いは、前者においては 'N' パケットが自然発生的に来るのに対して、'qOffsets' はホストのデバッガによって開始されたクエリーであるという点です。

0XX...

XX... は、ASCII データを 16 進符号化したものです。これは、プログラムが動作中で、デバッガが 'W'、'T'、等々を待ち続けなければならない状況ではいつでも発生し得ます。

以下のセット・パケットおよびクエリー・パケットは定義済みです。

|             |                     |                                                                                                                                                                                                                                         |
|-------------|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| カレント・スレッド   | qC                  | カレント・スレッドの ID を返します。                                                                                                                                                                                                                    |
|             | 応答 qCpid            | pid は 16 ビットのプロセス ID を 16 進符号化したものです。                                                                                                                                                                                                   |
|             | 応答*                 | その他の応答はいずれも古い PID ( プロセス ID ) を示唆します。                                                                                                                                                                                                   |
| すべてのスレッド ID | qfThreadInfo        |                                                                                                                                                                                                                                         |
|             | qsThreadInfo        | ターゲット ( OS ) から、すべてのアクティブなスレッドの ID のリストを取得します。アクティブなスレッドの数が多すぎて 1 つの応答パケットに入りきれない可能性があるため、このクエリーは対話的に行われます。スレッドの全リストを取得するためには、複数のクエリー / 応答シーケンスが必要になるかもしれません。シーケンスの最初のクエリーは qfThreadInfo クエリーです。また、シーケンス中の後続のクエリーは qsThreadInfo クエリーです。 |
|             |                     | 注: qL クエリーを置き換えます ( 後述 )。                                                                                                                                                                                                               |
|             | 応答 m<id>            | 単一のスレッド ID                                                                                                                                                                                                                              |
|             | 応答 m<id>,<id>...    | カンマで区切られたスレッド ID のリスト                                                                                                                                                                                                                   |
|             | 応答 1                | ( アルファベット小文字の「エル」 ) リストの終端を示します。                                                                                                                                                                                                        |
|             |                     | ターゲットは、個々のクエリーに対して、ビッグ・エンディアンの 16 進数であるスレッド ID ( 複数可 ) をカンマで区切ったリストを返します。GDB は、個々の応答に対して、( qs クエリーを使って ) さらにスレッド ID を求める要求を返します。これは、ターゲットが 1 ( アルファベット小文字の「エル」, 'last' を表わす ) を返すまで続きます。                                                |
| 追加のスレッド情報   | qThreadExtraInfo,id |                                                                                                                                                                                                                                         |

<id>はビッグ・エンディアンの 16 進数であるスレッド ID です。スレッド属性の表示可能な文字列による説明はターゲット OS から取得されます。この文字列には、そのスレッドに関して GDB からユーザに知らせたほうが良いとターゲット OS が考えたことであれば、何でも含まれている可能性があります。この文字列は、GDB の 'info threads' の表示結果の中に含まれます。追加のスレッド情報文字列として考えられる例としては、"Runnable"や"Blocked on Mutex"が挙げられます。

応答 XX...

XX... は ASCII データを 16 進符号化したもので、スレッドの属性に関する追加情報を含む表示可能な文字列から構成されます。

クエリー *LIST* またはクエリー *thread-LIST* (非推奨)

qLstartflagthreadcountnextthread

RTOS からスレッド情報を取得します。startflag (1 桁の 16 進数字) は、先頭のクエリーを示すときは 1、後続のクエリーを示すときは 0 です。threadcount (2 桁の 16 進数字) は、応答パケットに含まれるスレッド数の最大値です。nextthread (8 桁の 16 進数字) は、後続のクエリー (startflag が 0) に対するもので、応答内において argthread として返されます。

注: このクエリーは qfThreadInfo クエリーによって置き換えられます (既述)。

応答 qMcountdoneargthreadthread...

count (2 桁の 16 進数字) は返されたスレッドの数です。done (1 桁の 16 進数字) は、まだスレッドがあることを示す場合は 0、もうスレッドがないことを示す場合は 1 です。argthread<sup>1</sup> (8 桁の 16 進数字) は、要求パケットの中の nextthread です。thread... は、ターゲットから送られたスレッド ID のシーケンスです。<sup>2</sup>remote.c:parse\_threadlist\_response() を参照してください。

メモリ・ブロックの CRC の計算

qCRC:addr,length

応答 ENN

エラー (例えば、メモリ・フォルト)

応答 CCRC32

指定されたメモリ域の 32 ビット周期的冗長性チェック (cyclic redundancy check)

|                  |                             |                                                                                                                                                                                                                                   |
|------------------|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| セクション・オフセットのクエリー | <code>qOffsets</code>       | ダウンロードされたイメージを再配置する際にターゲットが使ったセクション・オフセットを取得します。注：応答には <code>Bss</code> オフセットが含まれていますが、 <i>GDB</i> はこれを無視し、代わりに <code>Bss</code> セクションに対して <code>Data</code> オフセットを適用します。                                                         |
|                  |                             | 応答 <code>Text=xxx;Data=yyy;Bss=zzz</code>                                                                                                                                                                                         |
| スレッド情報の要求        | <code>qPmodethreadid</code> | <i>threadid</i> に対応する情報を返します。 <i>mode</i> は 16 進符号化された 32 ビットのモードです。 <i>threadid</i> は 16 進符号化された 64 ビットのスレッド ID です。                                                                                                              |
|                  | 応答*                         | <code>remote.c:remote_unpack_thread_info_response()</code> を参照してください。                                                                                                                                                             |
| リモート・コマンド        | <code>qRcmd, COMMAND</code> | ( 16 進符号化された ) <i>COMMAND</i> は、実行するために、ローカルにあるインタプリタに渡されます。不正なコマンドは、出力文字列を通じて報告されるはずですが、最終的な結果パケットの前に、ターゲットがコンソール出力パケット <code>0OUTPUT</code> をいくつか応答する可能性もあります。実装者は、スタブのインタプリタへのアクセスを提供することには、セキュリティ上の問題がある可能性のあることに注意する必要があります。 |
|                  | 応答 OK                       | 出力を伴わないコマンド 応答。                                                                                                                                                                                                                   |
|                  | 応答 <i>OUTPUT</i>            | 16 進符号化された出力文字列 <i>OUTPUT</i> を伴うコマンド 応答。                                                                                                                                                                                         |
|                  | 応答 ENN                      | 要求が不正な形式であったことを示します。                                                                                                                                                                                                              |
|                  | 応答 ‘q’                      | ‘q’ <i>Rcmd</i> が認識できないときに返されます。                                                                                                                                                                                                  |

次の ‘g’/‘G’ パケットは以前は定義されていました。以下においては、いくつかの 32 ビット・レジスタが 64 ビットとして転送されています。これらのレジスタは、割り当てられた領域を埋めるためにゼロ拡張または符号拡張される必要があります。レジスタのバイトは、ターゲットのバイト・オーダーで転送されます。レジスタのバイトの中の 2 つの 4 ビットは、most-significant、least-significant の順に転送されます。

## MIPS32

すべてのレジスタは 32 ビットのサイズとして転送されます。転送の順序は、32 個の汎用レジスタ、*sr*、*lo*、*hi*、*bad*、*cause*、*pc*、32 個の浮動小数点レジスタ、*fsr*、*fir*、*fp* です。

## MIPS64

すべてのレジスタは ( srのような 32 ビットのレジスタも含めて ) 64 ビットのサイズとして転送されます。転送の順序は MIPS32 の場合と同じです。

ターゲットが再起動される場合のシーケンス例を以下に示します。再起動による直接の出力はまったくないことに注目してください。

```
<- R00
-> +
ターゲット再起動
<- ?
-> +
-> T001:1234123412341234
<- +
```

ターゲットを単一命令ごとにステップ実行する場合のシーケンス例を以下に示します。

```
<- G1445...
-> +
<- s
-> +
時間経過
-> T001:1234123412341234
<- +
<- g
-> +
-> 1455...
<- +
```

### 13.4.1.5 gdbserverプログラムの使用

`gdbserver` は、UNIX 系システム用の制御プログラムで、これにより、通常のデバッグ用スタブをリンクすることなく、`target remote` コマンドによって、ユーザ・プログラムをリモートの GDB に接続することができます。

`gdbserver` は、デバッグ用スタブに完全に取って代わるものではありません。`gdbserver` は、GDB が必要とするのと同様のオペレーティング・システムの機能を基本的には必要とするからです。実際、リモートの GDB と接続するために `gdbserver` を実行できるシステムであれば、GDB をローカルに実行することも可能でしょう。それでも、`gdbserver` は GDB と比較するとかなりサイズが小さいので、便利ことがあります。また、`gdbserver` の移植は GDB 全体の移植よりも簡単なので、`gdbserver` を使うことで、新しいシステムでの作業をより早く開始することができます。最後に、リアルタイム・システムをターゲットとする開発をしている場合、リアルタイムな操作に関わるトレードオフのために、例えばクロス・コンパイルなどによって、他のシステム上で可能な限り多くの開発作業を行ったほうが便利であるということがあられるでしょう。デバッグ作業に関しても、`gdbserver` を使うことでこれと同じような選択を行うことができます。GDB と `gdbserver` は、シリアル回線または TCP 接続を経由して、標準的な GDB リモート・シリアル・プロトコルによって通信します。

ターゲット・マシンでは：

デバッグしたいプログラムのコピーが 1 つ必要です。`gdbserver` はユーザ・プログラムのシンボル・テーブルを必要とはしませんので、スペースの節約が必要であれ



ば、プログラムをストリップすることができます。ホスト・システム上の GDB が、シンボルに関するすべての処理を実行します。

gdbserverを使うには、GDB との通信方法、ユーザ・プログラムの名前、ユーザ・プログラムへの引数を教えてやる必要があります。構文は、以下のとおりです。

```
target> gdbserver comm program [ args ... ]
```

*comm* は ( シリアル回線を使うための ) 装置名、あるいは、TCP のホスト名とポート番号です。例えば、`'foo.txt'` という引数を指定して Emacs をデバッグし、シリアル・ポート `'/dev/com1'` 経由で GDB と通信するには、以下のように実行します。

```
target> gdbserver /dev/com1 emacs foo.txt
```

gdbserverは、ホスト側の GDB が通信してくるのを受動的に待ちます。

シリアル回線の代わりに TCP 接続を使うには、以下のようにします。

```
target> gdbserver host:2345 emacs foo.txt
```

前の例との唯一の違いは第 1 引数です。これは、ホストの GDB と TCP によって通信することを指定しています。`'host:2345'` は、マシン `'host'` からローカルの TCP ポート 2345 への TCP 接続を gdbserver が期待していることを意味します ( 現在のバージョンでは、`'host'` の部分は無視されます )。ターゲット・システム上で既に使われている TCP ポートでなければ、任意の番号をポート番号として選択できます ( 例えば、23 は telnet に予約されています )<sup>3</sup>。ここで指定したのと同じポート番号を、ホスト上の GDB の `target remote` コマンドで使わなければなりません。

GDB のホスト・マシンでは :

GDB はシンボル情報、デバッグ情報を必要とするので、ストリップされていないユーザ・プログラムのコピーが必要です。通常どおり、第 1 引数にユーザ・プログラムのローカル・コピーの名前を指定して GDB を起動します ( シリアル回線の速度が 9600 bps 以外であれば、`'--baud'` オプションも必要になります )。その後、`target remote` コマンドによって gdbserver との通信を確立します。引数には、装置名 ( 通常は `'/dev/ttyb'` のようなシリアル装置 ) または、`host:PORT` という形式での TCP ポート記述子を指定します。例えば、

```
(gdb) target remote /dev/ttyb
```

では、シリアル回線 `'/dev/ttyb'` を介して gdbserver と通信します。また、

```
(gdb) target remote the-target:2345
```

では、ホスト `'the-target'` 上のポート 2345 に対する TCP 接続によって通信します。TCP 接続を使う場合には、`target remote` コマンドを実行する前に、gdbserver を起動しておかなければなりません。そうしないと、エラーになります。エラー・テキストの内容はホスト・システムによって異なりますが、通常は `'Connection refused'` のような内容です。

#### 13.4.1.6 gdbserve.nlmプログラムの使用

`gdbserve.nlm` は NetWare システム用の制御プログラムです。これによって、`target remote` コマンドでユーザ・プログラムをリモートの GDB に接続することができます。

<sup>3</sup> 原注 : 他のサービスによって使用されているポート番号を選択すると、gdbserver はエラー・メッセージを出力して終了します。

GDB と `gdbserve.nlm` は、標準の GDB リモート・シリアル・プロトコルを使って、シリアル回線経由で通信します。

ターゲット・マシンでは：

デバッグしたいプログラムのコピーが 1 つ必要です。 `gdbserve.nlm` はユーザ・プログラムのシンボル・テーブルを必要としないので、スペースの節約が必要であれば、プログラムをストリップすることができます。ホスト・システム上の GDB が、シンボルに関わるすべての処理を実行します。

`gdbserve.nlm` を使うには、GDB との通信方法、ユーザ・プログラムの名前、ユーザ・プログラムの引数を教えてやる必要があります。構文は、以下のとおりです。

```
load gdbserve [ BOARD=board ] [ PORT=port ]
               [ BAUD=baud ] program [ args ... ]
```

*board* と *port* がシリアル回線を指定します。 *baud* は接続に使われるボーレートを指定します。 *port* と *node* のデフォルト値は 0、 *baud* のデフォルト値は 9600 bps です。

例えば、 `'foo.txt'` という引数を指定して Emacs をデバッグし、シリアル・ポート番号 2、ボード 1 を経由して 19200 bps の接続で GDB と通信するには、以下のように実行します。

```
load gdbserve BOARD=1 PORT=2 BAUD=19200 emacs foo.txt
```

GDB のホスト・マシンでは：

GDB はシンボル情報、デバッグ情報を必要とするので、ストリップされていないユーザ・プログラムのコピーが必要です。通常どおり、第 1 引数にユーザ・プログラムのローカル・コピーの名前を指定して GDB を起動します (シリアル回線の速度が 9600 bps 以外であれば、 `'--baud'` オプションも必要になります)。その後、 `target remote` コマンドによって `gdbserve.nlm` との通信を確立します。引数には、装置名 (通常は `'/dev/ttyb'` のようなシリアル装置) を指定します。例えば、

```
(gdb) target remote /dev/ttyb
```

は、シリアル回線 `'/dev/ttyb'` を経由して `gdbserve.nlm` と通信します。

### 13.5 カーネル・オブジェクト表示

いくつかのターゲットでは、カーネル・オブジェクトの表示がサポートされています。GDB は、この機能を使うことで、オペレーティング・システムと特別に通信を行い、 `mutex` やその他の同期オブジェクトのようなオペレーティング・システム階層におけるオブジェクトに関する情報を表示することができます。表示可能なオブジェクトは OS ごとに決まっています。

オペレーティング・システムを指定するには `set os` コマンドを使います。これにより GDB に対して、初期化するべきカーネル・オブジェクト表示モジュールが指示されます。

```
(gdb) set os cisco
```

`set os` が成功すると、GDB はオペレーティング・システムに関していくつかの情報を表示し、ターゲットに関するクエリー (問い合わせ) に使うことのできる新しい `info` コマンドが生成されます。 `info` コマンドには、オペレーティング・システムをもとに名前が付与されます。

```
(gdb) info cisco
List of Cisco Kernel Objects
Object      Description
any         Any and all objects
```

さらにサブコマンドを使うことによって、カーネルが認識している特定のオブジェクトに関して問い合わせることができます。

現時点では、あるオペレーティング・システムがサポートされているかどうかを判定するには、実際に試してみるよりほかに方法がありません。



## 14 コンフィギュレーション固有の情報

GDB コマンドのほとんどすべては、このデバッガのネイティブ版、クロス版のすべてにおいて利用可能ですが、これには例外もあります。この章では、特定のコンフィギュレーションにおいてのみ利用可能なことについて説明します。

コンフィギュレーションには 3 つの主要なカテゴリがあります。ネイティブ・コンフィギュレーションにおいては、ホストとターゲットは同一です。組み込みオペレーティング・システムのコンフィギュレーションにおいても、いくつかの異なるプロセッサ・アーキテクチャにおいては、通常はホストとターゲットが同一です。一方、組み込みプロセッサ ( ボード ) のコンフィギュレーションでは、これらは極めて異なるものとなります。

### 14.1 ネイティブ

このセクションでは、特定のネイティブ・コンフィギュレーションに固有な詳細について説明します。

#### 14.1.1 HP-UX

HP-UX システム上においてドル記号で始まる関数名や変数名を指定すると、GDB は、コンパニエンス変数を探す前に、ユーザ名やシステム名を探します。

#### 14.1.2 SVR4 プロセス情報

SVR4 の多くのバージョンは、`‘/proc’` と呼ばれる便利な機能を提供しています。これは、ファイル・システム関連のサブルーチンを使用して、実行中プロセスのイメージを調べるのに使用することができます。GDB が、この機能を持つオペレーティング・システム用に構成されていれば、`info proc` コマンドを使用することで、ユーザ・プログラムを実行しているプロセスに関するいくつかの情報を知ることができます。`info proc` は、`procfs` コードを持つ SVR4 システム上でのみ機能します。これには例えば、OSF/1 ( Digital Unix )、Solaris、Irix、Unixware が含まれますが、HP-UX や Linux は含まれません。

`info proc` プロセスに関して入手可能な情報を要約して出力します。

`info proc mappings`

プログラムがアクセスすることのできるアドレス範囲に関する情報を表示します。出力情報には、それぞれのアドレス範囲に対してユーザ・プログラムが持つ読み取り権、書き込み権、実行権の情報が含まれます。

`info proc times`

ユーザ・プログラムおよびその子 ( プロセス ) の起動時刻、ユーザ・レベルの CPU 消費時間、システム・レベルの CPU 消費時間を表示します。

`info proc id`

ユーザ・プログラムと関連付けられたプロセスの ID 情報を表示します。ユーザ・プログラムのプロセス ID、親 ( プロセス ) のプロセス ID、プロセス・グループ ID、セッション ID を出力します。

`info proc status`

プロセスの状態に関する一般的な情報を出力します。プロセスが停止している場合は停止した理由、( シグナルを受信した場合には ) 受信したシグナルが出力情報に含まれます。

```
info proc all
```

プロセスに関する上記の情報をすべて表示します。

## 14.2 組み込みオペレーティング・システム

このセクションでは、いくつかの異なるアーキテクチャに対して利用可能な組み込みオペレーティング・システムのデバッグに関するコンフィギュレーションについて説明します。

GDB には、さまざまなリアルタイム・オペレーティング・システム上で動作するプログラムをデバッグする機能が含まれています。

### 14.2.1 VxWorks における GDB の使用

```
target vxworks machinename
```

TCP/IP で接続された VxWorks システムです。引数 *machinename* は、ターゲット・システムのマシン名または IP アドレスです。

VxWorks で load コマンドを実行すると、*filename* で指定される実行ファイルがカレントなターゲット・システム上で動的にリンクされ、シンボルが GDB に追加されます。

開発者は、GDB を使用することによって、ネットワークに接続された VxWorks ターゲット上のタスクを、UNIX のホストから起動してデバッグすることができます。VxWorks シェルから起動され、既に実行中の状態のタスクをデバッグすることもできます。GDB は、UNIX ホスト上で実行されるコードと VxWorks ターゲット上で実行されるコードの両方を使います。gdb プログラムは、UNIX ホスト上にインストールされて実行されます（ホスト上のプログラムをデバッグするのに使う GDB と区別するために、vxgdb という名前でインストールされることもあります）。

```
VxWorks-timeout args
```

すべての VxWorks ベースのターゲットが、vxworks-timeout オプションをサポートするようになりました。このオプションはユーザによってセットされるもので、*args* は、GDB が RPC の応答を待つ秒数を表わします。実際の VxWorks ターゲットが速度の遅いソフトウェア・シミュレータであったり、帯域の小さいネットワーク回線を介して遠距離にある場合などに使うとよいでしょう。

VxWorks との接続に関する以下の情報は、このマニュアルの作成時における最新の情報です。新しくリリースされた VxWorks では、手順が変更されているかもしれません。

VxWorks 上で GDB を使うためには、VxWorks カーネルを再構築して、VxWorks ライブラリ 'rdb.a' の中のリモート・デバッグ用のインターフェイス・ルーチンを組み込む必要があります。そのためには、VxWorks のコンフィギュレーション・ファイル 'configAll.h' の中で INCLUDE\_RDB を定義して、VxWorks カーネルを再構築します。この結果として生成されるカーネルには 'rdb.a' が組み込まれ、VxWorks の起動時にソース・デバッグ用のタスク tRdbTask が起動されます。VxWorks の構成や再構築に関する詳細については、製造元のマニュアルを参照してください。

VxWorks システム・イメージへの 'rdb.a' の組み込みが終わり、UNIX の実行ファイル・サーチ・パスに GDB の存在するパスを加えれば、GDB を実行するための準備は完了です。UNIX ホストから gdb（インストールの方法によっては vxgdb）を実行します。

GDB が起動されて、以下のプロンプトを表示します。

```
(vxgdb)
```

### 14.2.1.1 VxWorks への接続

GDB の `target` コマンドによって、ネットワーク上の VxWorks ターゲットに接続します。tt というホスト名を持つターゲットに接続するには、以下のようにします。

```
(vxgdb) target vxworks tt
```

GDB は以下のようなメッセージを表示します。

```
Attaching remote machine across net...
Connected to tt.
```

続いて GDB は、最後に VxWorks ターゲットが起動されたときより後にロードされたオブジェクト・モジュールのシンボル・テーブルを読み込もうと試みます。GDB は、コマンドのサーチ・パス (セクション 4.4 [ユーザ・プログラムの環境], ページ 23 参照) に含まれているディレクトリを探索することによって、これらのファイルを見つけます。オブジェクト・ファイルを見つけない場合には、以下のようなメッセージを表示します。

```
prog.o: No such file or directory.
```

このような場合には、GDB の `path` コマンドによって適切なディレクトリを検索パスに加えてから、再度 `target` コマンドを実行します。

### 14.2.1.2 VxWorks ダウンロード

VxWorks ターゲットに接続済みの状態で、まだロードされていないオブジェクトをデバッグしたい場合には、GDB の `load` コマンドを使って UNIX から VxWorks へ追加的にファイルをダウンロードすることができます。`load` コマンドの引数として指定されたオブジェクト・ファイルは、実際には 2 回オープンされます。まず、コードをダウンロードするために VxWorks ターゲットによってオープンされ、次にシンボル・テーブルを読み込むために GDB によってオープンされます。2 つのシステム上のカレントな作業ディレクトリが異なると、問題が発生します。両方のシステムが同一のファイル・システムを NFS マウントしているのであれば、絶対パスを使うことで問題を回避することができます。そうでない場合は、両方のシステム上で、オブジェクト・ファイルが存在するディレクトリを作業ディレクトリにして、パスを一切使わずにファイル名だけでファイルを参照するのが、最も簡単でしょう。例えば、プログラム `'prog.o'` が、VxWorks では `'vxpath/vw/demo/rdb'` に存在し、ホストでは `'hostpath/vw/demo/rdb'` に存在するとしましょう。このプログラムをロードするには、VxWorks 上で以下のように実行します。

```
-> cd "vxpath/vw/demo/rdb"
```

GDB 上では、以下のように実行します。

```
(vxgdb) cd hostpath/vw/demo/rdb
(vxgdb) load prog.o
```

GDB は次のような応答を表示します。

```
Reading symbol data from wherever/vw/demo/rdb/prog.o... done.
```

ソース・ファイルを編集して再コンパイルした後に、`load` コマンドを使ってオブジェクト・モジュールを再ロードすることもできます。ただし、これを行うと、GDB はその時点で定義されているすべてのブレイクポイント、自動表示設定、コンビニエンス変数を削除し、値履歴を初期化してしまいますので、注意してください (これは、ターゲット・システムのシンボル・テーブルを参照するデバッガのデータ構造の完全性を保つために必要です)。

### 14.2.1.3 タスクの実行

以下のように `attach` コマンドを使うことで、既存のタスクにアタッチすることも可能です。

(vxgdb) attach *task*

*task* は、VxWorks の 16 進数のタスク ID です。アタッチするときに、タスクは実行中であってもサスペンドされていても構いません。実行中であったタスクは、アタッチされたときにサスペンドされます。

## 14.3 組み込みプロセッサ

このセクションでは、特定の組み込みコンフィギュレーションに固有な詳細を説明します。

### 14.3.1 組み込み AMD A29K

target adapt *dev*

モニタを A29K に適応させます。

target amd-eb *dev speed PROG*

シリアル回線により接続されている、リモートの PC に組み込まれた AMD EB29K ボードです。target remote の場合と同様、*dev* はシリアル装置です。*speed* によって回線速度を指定することができます。*PROG* は、デバッグ対象となるプログラムを PC 上の DOS から見た場合の名前です。セクション 14.3.2 [AMD29K の EBMON プロトコル], ページ 142 を参照してください。

#### 14.3.1.1 A29K UDI

GDB は、a29k プロセッサ・ファミリをデバッグするための AMD UDI ( Universal Debugger Interface ) プロトコルをサポートしています。MiniMON モニタを実行する AMD ターゲットという構成を使うには、AMD 社から無料で入手可能な MONTIP プログラムが必要になります。また、AMD 社から入手可能な UDI 準拠の a29k シミュレータ・プログラム ISSTIP とともに GDB を使うこともできます。

target udi *keyword*

リモートの a29k ボードまたはシミュレータへの UDI インターフェイスを選択します。*keyword* は、AMD コンフィギュレーション・ファイル 'udi\_soc' 内のエントリです。このファイルには、a29k ターゲットに接続するときに使われるパラメータを指定するキーワード・エントリが含まれます。'udi\_soc' ファイルが作業ディレクトリにない場合には、環境変数 'UDICONF' にそのパス名を設定しなければなりません。

### 14.3.2 AMD29K の EBMON プロトコル

AMD 社は、PC 組み込み用の 29K 開発ボードを、DOS 上で動作する EBMON というモニタ・プログラムとともに配布しています。この開発システムは、省略して EB29K と呼ばれます。UNIX システム上の GDB を使って EB29K ボード上でプログラムを実行するには、まず ( EB29K を組み込んだ ) PC と UNIX システムのシリアル・ポートの間をシリアル回線で接続しなければなりません。以下の節では、PC の 'COM1' ポートと UNIX システムの '/dev/ttya' との間をケーブルで接続してあるものと仮定します。



### 14.3.2.1 通信セットアップ

PC 上の DOS で以下のように実行することによって、PC のポートをセットアップします。

```
C:\> MODE com1:9600,n,8,1,none
```

MS DOS 4.0 上で実行されているこの例では、PC ポートを通信速度 9600 bps、パリティ・ビットなし、データ・ビット数 8、ストップ・ビット数 1、リトライなしに設定しています。UNIX 側を設定する際には、同一の通信パラメータを使わなければなりません。

シリアル回線の UNIX 側に PC の制御権を与えるには、DOS コンソール上で以下のように実行します。

```
C:\> CTTY com1
```

(後に、DOS コンソールに制御を戻したいときには、CTTY con コマンドを使うことができます。ただし、制御権を持っている装置からこのコマンドを送信する必要があります。ここでの例では、‘COM1’に接続されているシリアル回線を通して送信することになります)。

UNIX のホストからは、PC と通信するのに tip や cu のような通信プログラムを使います。以下に例を示します。

```
cu -s 9600 -l /dev/ttya
```

ここで示されている cu オプションはそれぞれ、使用する回線速度とシリアル・ポートを指定しています。tip コマンドを使った場合は、コマンドラインは以下のようなものになるでしょう。

```
tip -9600 /dev/ttya
```

ここで tip への引数として指定した ‘/dev/ttya’ の部分には、システムによって異なる名前を指定する必要があるかもしれません。使用するポートを含む通信パラメータは、remote 記述ファイルにおいて tip コマンドへの引数と関連付けられます。通常このファイルは、システム・テーブル ‘/etc/remote’ です。

tip 接続または cu 接続を使用して DOS の作業ディレクトリを 29K プログラムが存在するディレクトリに変更し、PC プログラム EBMON (AMD 社からボードとともに提供される EB29K 制御プログラム) を起動します。以下に示す例によく似た、EBMON プロンプト ‘#’ で終わる EBMON の初期画面が表示されるはずです。

```
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

```

```
# ~.
```

続いて、cuプログラムまたはtipプログラムを終了させます（上の例では、EBMONプロンプトにおいて~.を入力することで終了させています）。EBMONは、GDBが制御権を獲得できる状態で、実行を継続します。

この例では、PCとUNIXシステムの両方に同一の29Kプログラムが確実に存在するようにするのに、おそらく最も便利であろうと思われる方法を使うことを仮定しました。それは、PC/NFSによる接続で、PCのG:ドライブをUNIXホストのファイル・システムの1つとする方法です。PC/NFS、あるいは、2つのシステム間を接続する類似の方法がない場合、フロッピー・ディスクによる転送など、UNIXシステムからPCへ29Kプログラムを転送するための他の手段を準備する必要があります。GDBは、シリアル回線経由で29Kプログラムをダウンロードすることはしません。

### 14.3.2.2 EB29K クロス・デバッグ

最後に、UNIXシステム上の29Kプログラムが存在するディレクトリにcdコマンドによって移動して、GDBを起動します。引数には、29Kプログラムの名前を指定します。

```

cd /usr/joe/work29k
gdb myfoo

```

これでtargetコマンドが使えるようになります。

```
target amd-eb /dev/ttya 9600 MYF00
```

この例では、ユーザ・プログラムは‘myfoo’と呼ばれるファイルであると仮定しています。target amd-ebに対して最後の引数として指定するファイル名は、DOS上でのプログラム名でなければならない点に注意してください。この例では単にMYF00となっていますが、DOSのパス名を含むこともできますし、転送メカニズムによっては、UNIX側での名前とは似ても似つかないものになることもあるでしょう。

ここまできると、好きなようにブレイクポイントを設定することができます。29Kボード上でのプログラムの実行を監視する準備が整えば、GDBのrunコマンドを使います。

リモート・プログラムのデバッグを停止するには、GDBのdetachコマンドを使います。

PCの制御をPCコンソールに戻すには、GDBセッションが終了した後に、EBMONにアタッチするために、もう一度tipまたはcuを使います。その後、qコマンドによってEBMONをシャットダウンし、DOSのコマンドライン・インタプリタに制御を戻します。CTTY conと入力して、入力されたコマンドがメインのDOSコンソールによって受け取られるようにし、~.を入力してtipまたはcuを終了させます。

### 14.3.2.3 リモート・ログ

target amd-ebコマンドは、接続に関わる問題のデバッグを支援するため、カレントな作業ディレクトリに‘eb.log’というファイルを作成します。‘eb.log’は、EBMONに送信されたコマンド

のエコーを含む、EBMONからのすべての出力を記録します。別のウィンドウ内でこのファイルに対して `'tail -f'` を実行すると、EBMONに関わる問題や PC 側での予期せぬイベントを理解する助けになることがよくあります。

### 14.3.3 ARM

`target rdi dev`

ADP プロトコルに対する RDI ライブラリ・インターフェイスを経由した ARM Angel モニタです。Angel モニタを動作させるボード、EmbeddedICE JTAG デバッグ装置のいずれと通信する場合であっても、このターゲットを使うことができます。

`target rdp dev`

ARM Demon モニタです。

### 14.3.4 日立 H8/300

`target hms dev`

ユーザのホストにシリアル回線で接続された日立の SH、H8/300、H8/500 ボードです。特別なコマンドである `device` と `speed` によって、使用されるシリアル回線と通信速度を制御します。

`target e7000 dev`

日立 H8、SH 用の E7000 エミュレータです。

`target sh3 dev`

`target sh3e dev`

日立 SH-3、SH-3E ターゲット・システムです。

日立の SH、H8/300、H8/500 ボードに対するリモート・デバッグを選択すると、`load` コマンドはユーザ・プログラムを日立ボードにダウンロードし、(`file` コマンドと同様) ユーザのホスト・マシン上の GDB のカレントな実行ファイル・ターゲットとしてオープンします。

日立の SH、H8/300、H8/500 と通信するためには、GDB は以下の情報を知っている必要があります。

1. ユーザは、日立マイクロプロセッサへのリモート・デバッグ用インターフェイスである `'target hms'` と、日立 SH や日立 300H のインサーキット・エミュレータである `'target e7000'` のどちらを使用したいかということ (GDB が日立 SH、H8/300、H8/500 用に特に構成されている場合には、`'target hms'` がデフォルトです)
2. ホストと日立ボードを接続しているシリアル装置 (デフォルトは、ホスト上で利用できる最初のシリアル装置です)
3. シリアル装置で使用する速度

#### 14.3.4.1 日立ボードへの接続

シリアル装置を明示的に指定する必要があるれば、そのための専用コマンドである、GDB の `'device port'` コマンドを使用します。`port` のデフォルトは、ホスト上で最初に利用可能なポートです。これは UNIX ホスト上でのみ必要であり、そこでは典型的には `'/dev/ttya'` のような名前になります。

GDBには、通信速度を設定するためのもう1つの専用コマンド ‘speed bps’があります。このコマンドもまた UNIX ホストからのみ使用されるものです。DOS ホストでは通常どおり、GDB からではなく DOS の mode コマンドによって回線速度を設定します（例えば、9600 bps の接続を確立するには `mode com2:9600,n,8,1,p` のように実行します）。

‘device’コマンドと ‘speed’コマンドは、日立マイクロプロセッサ・プログラムのデバッグに UNIX ホストを使う場合のみ利用可能です。DOS ホストを使う場合、GDB は、PC のシリアル・ポート経由で開発ボードと通信するのに、`asynctsr`と呼ばれる補助的な常駐プログラムに依存します。DOS 側でシリアル・ポートの設定をする場合にも、DOS の mode コマンドを使わなければなりません。

以下のサンプル・セッションは、GDB の制御のもとに、H8/300 上においてプログラムを起動するのに必要とされる方法を示したものです。この例では、‘t.x’という名前の H8/300 サンプル・プログラムを使用します。手順自体は、日立 SH や H8/500 でも同様です。

まず最初に、開発ボードを取り付けます。この例では、シリアル・ポート COM2 に接続したボードを使用します。異なるシリアル・ポートを使用するのであれば、mode コマンドの引数の中の名前を置き換えます。デバッガにより使用される補助的な通信プログラムである `asynctsr` を呼び出すときには、シリアル・ポート名の数字の部分だけを渡します。例えば、以下の例における ‘`asynctsr 2`’ は、COM2 において `asynctsr` を実行します。

```
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
```

注意：PC-NFS には、`asynctsr` と衝突するようなバグのあることが判明しています。DOS ホスト上で PC-NFS を実行しているのであれば、開発ボードを制御するのに `asynctsr` を使用するためには、PC-NFS を停止するか、場合によっては、PC-NFS が実行されないようにしてホストを起動し直す必要があるかもしれません。

シリアル通信がセットアップされて、開発ボードが接続されれば、GDB を起動することができます。プログラムの名前を引数にして `gdb` を呼び出します。GDB は、通常どおり、‘(gdb)’ というプロンプトによって入力待ちになります。デバッグ・セッションを開始するには、2 つの特別なコマンドを使用します。日立ボードに対するクロス・デバッグを指定するには ‘`target hms`’ コマンドを使用します。また、ボードにプログラムをダウンロードするには `load` コマンドを使用します。`load` コマンドは、プログラムのセクション名を表示し、さらに、2K のデータをダウンロードするたびに ‘\*’ を表示します。（GDB の持っているシンボルや実行ファイルに関するデータをダウンロードすることなく最新の状態にしたいのであれば、GDB の `file` コマンドや `symbol-file` コマンドを使用します。これらのコマンドと `load` コマンド自体については、セクション 12.1 [ファイル指定するコマンド]、ページ 109 に説明されています。）

```
(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 *
```

この時点において、プログラムを実行したりデバッグしたりする準備が整いました。ここからは、普通の GDB コマンドはすべて使用することができます。break コマンドによるブレイクポイントのセット、run コマンドによるプログラムの実行開始、print コマンドや x コマンドによるデータの表示、ブレイクポイントにおける停止後の continue コマンドによる実行再開が可能です。GDB のコマンドに関して詳細を知るには、いつでも help コマンドを使用することができます。しかし、開発ボード上では、オペレーティング・システムの機能は利用できないということを忘れないでください。例えば、プログラムがハングしても、割り込みを送信することはできません。もちろん、RESET ボタンを押すことはできます！

開発ボード上で以下のことを行うには、RESET ボタンを使用してください。

- プログラムに割り込みをかける。(DOS ホスト上で `ctl-C` を使用しないでください。DOS ホストから開発ボードに対して割り込みシグナルを送る方法はありません。)
- プログラムが正常終了した後に、GDB のコマンド・プロンプトに戻る。通信プロトコルは、GDB がプログラムの完了を検出するための方法をこれ以外に提供していません。

どちらの場合でも、開発ボード上での RESET は、GDB からはプログラムの「正常終了」と見えます。

#### 14.3.4.2 E7000 インサーキット・エミュレータの使用

E7000 インサーキット・エミュレータを使って、日立 SH または H8/300H 用のコードを開発することができます。‘target e7000’ コマンドを以下のいずれかの形式で使って、GDB を E7000 に接続します。

```
target e7000 port speed
```

E7000 がシリアル・ポートに接続されている場合は、この形式を使ってください。引数 *port* が、使用するシリアル・ポートを指定します (例えば、‘com2’)。3 番目の引数は、秒あたりのビット数による回線速度です (例えば、‘9600’)。

```
target e7000 hostname
```

E7000 が TCP/IP ネットワーク上のホストとしてインストールされている場合、ホスト名だけを指定することもできます。GDB は接続に telnet を使います。

#### 14.3.4.3 日立マイクロプロセッサ用の特別な GDB コマンド

いくつかの GDB コマンドは H8/300 においてのみ利用可能です。

```
set machine h8300
set machine h8300h
```

‘set machine’ コマンドによって、2 種類の H8/300 アーキテクチャのどちらか一方にあわせて GDB を調整します。‘show machine’ コマンドによって、現在有効なアーキテクチャを調べることができます。

### 14.3.5 H8/500

```
set memory mod
show memory
```

‘set memory’コマンドによって、使用中の H8/500 メモリ・モデル (*mod*) を指定します。‘show memory’コマンドによって、現在有効なメモリ・モデルを調べます。*mod* に指定可能な値は、small、big、medium、compactのいずれかです。

### 14.3.6 Intel i960

```
target mon960 dev
```

Intel i960 用の MON960 モニタです。

```
target nindy devicename
```

Nindy Monitor により制御される Intel 960 ボードです。*devicename* は、接続に使用するシリアル装置の名前です。例えば ‘/dev/ttya’ です。

Nindy は、Intel 960 ターゲット・システム用の ROM Monitor プログラムです。Nindy を使ってリモートの Intel 960 を制御するよう GDB が構成されている場合、いくつかの方法によって GDB に 960 との接続方法を教えることができます。

- シリアル・ポート、Nindy プロトコルのバージョン、通信スピードを指定するコマンドライン・オプションによる方法
- 起動時のプロンプトに答える方法
- GDB セッション中の任意の時点で target コマンドを使う方法 (セクション 13.2 [ターゲットを管理するコマンド], ページ 115 を参照してください)

Intel 960 ボードの Nindy インターフェイスでは、load コマンドは *filename* で指定されるファイルを 960 側にダウンロードし、そのシンボルを GDB に追加します。

#### 14.3.6.1 Nindy 使用時の起動方法

コマンドライン・オプションを一切使わずに gdb を起動すると、通常の GDB プロンプトが表示される前に、使用するシリアル・ポートを指定するよう促されます。

```
Attach /dev/ttyNN -- specify NN, or "quit" to quit:
```

このプロンプトに対して、使いたいシリアル・ポートを示す (‘/dev/tty’ の後ろの) サフィックスを入力します。もしそうしたいのであれば、プロンプトに空行で答えることによって、Nindy との接続を確立せずに起動することもできます。この場合、後に Nindy と接続したいときには target コマンドを使います (セクション 13.2 [ターゲットを管理するコマンド], ページ 115 参照)。

#### 14.3.6.2 Nindy 用のオプション

接続された Nindy-960 ボードとの GDB セッションを開始するための起動オプションを以下に示します。

```
-r port      ターゲット・システムとの接続に使用されるシリアル・インターフェイスのシリアル・ポート名を指定します。このオプションは、GDB が Intel 960 ターゲット・
```

アーキテクチャ用に構成されているときのみ利用可能です。 *port* は、完全なパス名 (例: `‘-r /dev/ttya’`)、`‘/dev’`配下のデバイス名 (例: `‘-r ttya’`)、tty固有の一意的サフィックス (例: `‘-r a’`) のいずれによっても指定することができます。

`-0` (ゼロではなく、英大文字の O です)。GDB がターゲット・システムと接続する際に、「古い」Nindy モニタ・プロトコルを使用すべきであることを指定します。このオプションは、GDB が Intel 960 ターゲット・アーキテクチャ用に構成されているときのみ利用可能です。

注意: `‘-0’`を指定したにもかかわらず、実際にはより新しいプロトコルを期待しているターゲット・システムに接続しようとした場合、接続は失敗します。この失敗は、あたかも通信速度の不一致が原因であるかのように見えてしまいます。GDB は、異なる回線速度によって再接続を繰り返し試みます。割り込みによって、この処理を中断させることができます。

`-brk` 接続する前に Nindy ターゲットをリセットするために、ターゲット・システムに対して最初に BREAK 信号を送信するよう、GDB に対して指定します。

注意: 多くのターゲット・システムは、このオプションが必要とするハードウェアを備えていません。このオプションが機能するボードは少数です。

標準の `‘-b’` オプションが、シリアル・ポート上で使用される回線速度を制御します。

### 14.3.6.3 Nindy reset コマンド

`reset` ターゲットが Nindy である場合、このコマンドは BREAK 信号をリモートのターゲット・システムに送信します。これは、BREAK 信号を受信したときにハード・リセット (または、その他の興味深いアクション) を実行する回路がターゲットに備わっている場合にのみ役に立ちます。

### 14.3.7 三菱 M32R/D

`target m32r dev`  
三菱 M32R/D ROM モニタです。

### 14.3.8 M68k

Motorola m68k 用のコンフィギュレーションには ColdFire サポート、および、以下の ROM モニタに対する `target` コマンドが含まれています。

`target abug dev`  
M68K 用の ABug ROM モニタです。

`target cpu32bug dev`  
CPU32 (M68K) ボード上で動作する CPU32BUG モニタです。

`target dbug dev`  
Motorola ColdFire 用の dBUG ROM モニタです。

`target est dev`  
CPU32 (M68K) ボード上で動作する EST-300 ICE モニタです。

```
target rom68k dev
```

M68K IDP ボード上で動作する ROM 68K モニタです。

GDB が `m68*-ericsson-*` によって構成されている場合、これらの代わりに、特別な `target` コマンドを 1 つだけ持つことになります。<sup>1</sup>

```
target es1800 dev
```

M68K 用の ES-1800 エミュレータです。

### 14.3.9 M88K

```
target bug dev
```

MVME187 (m88k) ボード上で動作する BUG モニタです。

### 14.3.10 組み込み MIPS

GDB は、MIPS のリモート・デバッグ用のプロトコルを使って、シリアル回線に接続された MIPS ボードと通信することができます。これは、GDB の構成時に `configure` に対して '`--target=mips-idt-ecoff`' オプションを指定することによって、利用することができます。ターゲット・ボードとの接続を指定するには、以下の GDB コマンドを使用します。

```
target mips port
```

ボード上でプログラムを実行するには、引数にユーザ・プログラムの名前を指定して `gdb` を起動します。ボードに接続するには、'`target mips port`' コマンドを使用します。`port` は、ボードに接続されているシリアル・ポートの名前です。プログラムがまだボードにダウンロードされていないのであれば、`load` コマンドを使ってダウンロードすることができます。その後、通常利用できるすべての GDB コマンドを使うことができます。

例えば以下の手順では、デバッガを使うことによって、シリアル・ポートを経由してターゲット・ボードに接続した後に、`prog` と呼ばれるプログラムをロードして実行しています。

```
host$ gdb prog
GDB is free software and ...
(gdb) target mips /dev/ttyb
(gdb) load prog
(gdb) run
```

```
target mips hostname:portnumber
```

GDB のホスト構成によっては、'`hostname:portnumber`' という構文を使うことで、シリアル・ポートの代わりに (例えば、端末多重化装置によって管理されているシリアル回線への) TCP 接続を指定することができます。

```
target pmon port
```

PMON ROM モニタです。

```
target ddb port
```

NEC が提供している、Vr4300 用 PMON の DDB 版です。

---

<sup>1</sup> 訳注: 原文ではここに `es1800` と `rombug` の 2 つのターゲットに関する記述がありますが、`rombug` のほうは無意味と思われるので削除しました。



```
target lsi port
    PMON の LSI 版です。
```

```
target r3900 dev
    東芝 R3900 Mips 用の Densan DVE-R3900 ROM モニタです。
```

```
target array dev
    Array Tech LSI33K RAID コントローラ・ボードです。
```

GDB は、MIPS ターゲットに対して、以下の特別なコマンドもサポートしています。

```
set processor args
show processor
```

プロセッサの種類に固有のレジスタにアクセスしたい場合には、`set processor` コマンドを使って MIPS プロセッサの種類を設定します。例えば、`set processor r3041` は、3041 チップで有効な CPO レジスタを使うよう GDB に対して通知します。GDB が使っている MIPS プロセッサの種類を知るには、`show processor` コマンドを使います。GDB が使っているレジスタを知るには、`info reg` コマンドを使います。

```
set mipsfpu double
set mipsfpu single
set mipsfpu none
show mipsfpu
```

MIPS 浮動小数点コプロセッサをサポートしないターゲット・ボードを使う場合は、`'set mipsfpu none'` コマンドを使う必要があります (このようなことが必要な場合には、GDB 初期化ファイルの中にそのコマンドを入れてしまってもよいでしょう)。これによって、浮動小数値を返す関数の戻り値を見つける方法を GDB に知らせます。またこれにより、ボード上で関数を呼び出すときに、GDB は浮動小数点レジスタの内容を退避する必要がなくなります。R4650 プロセッサ上にあるような、単精度浮動小数だけをサポートする浮動小数点コプロセッサを使っている場合には、`'set mipsfpu single'` コマンドを使います。デフォルトの倍精度浮動小数点コプロセッサは、`'set mipsfpu double'` によって選択することができます。

以前のバージョンでは、有効な選択肢は、倍精度浮動小数点コプロセッサを使う設定と浮動小数点コプロセッサを使わない設定だけでした。したがって、`'set mipsfpu on'` で倍精度浮動小数コプロセッサが選択され、`'set mipsfpu off'` で浮動小数点コプロセッサを使わないという設定が選択されていました。

他の場合と同様、`mipsfpu` 変数に関する設定は、`'show mipsfpu'` によって問い合わせることができます。

```
set remotedebug n
show remotedebug
```

`remotedebug` 変数を設定することによって、ボードとの通信に関するいくつかのデバッグ用の情報を見ることができます。`'set remotedebug 1'` によって値 1 を設定すると、すべてのパケットが表示されます。値を 2 に設定すると、すべての文字が表示されます。`'show remotedebug'` コマンドによって、いつでも現在の設定値を調べることができます。

```

set timeout seconds
set retransmit-timeout seconds
show timeout
show retransmit-timeout

```

MIPS リモート・プロトコルにおけるパケット待ちの状態でのタイムアウト時間を、`set timeout seconds` コマンドで制御することができます。デフォルトは 5 秒です。同様に、パケットに対する確認 (ACK) を待っている状態でのタイムアウト時間を、`set retransmit-timeout seconds` コマンドで制御することができます。デフォルトは 3 秒です。それぞれの値を `show timeout` と `show retransmit-timeout` で調べることができます (どちらのコマンドも、GDB が ‘`--target=mips-idt-ecoff`’ 用に構成されている場合のみ使用可能です)。

`set timeout` で設定されたタイムアウト時間は、ユーザ・プログラムが停止するのを GDB が待っている間は適用されません。この場合には、GDB は永遠に待ち続けます。これは、停止するまでにプログラムがどの程度長く実行を継続するのかわかる方法がないからです。

### 14.3.11 PowerPC

```

target dink32 dev
    DINK32 ROM モニタです。

target ppcbug dev
target ppcbug1 dev
    PowerPC 用の PPCBUG ROM モニタです。

target sds dev
    ( Motorola の ADS などの ) PowerPC ボード上で動作する SDS モニタです。

```

### 14.3.12 組み込み HP PA

```

target op50n dev
    OKI HPPA ボード上で動作する OP50N モニタです。

target w89k dev
    Winbond HPPA ボード上で動作する W89K モニタです。

```

### 14.3.13 日立 SH

```

target hms dev
    ユーザのホストにシリアル回線で接続された日立の SH ボードです。特別なコマンドである device と speed によって、使用されるシリアル回線と通信速度を制御します。

target e7000 dev
    日立 SH 用の E7000 エミュレータです。

target sh3 dev
target sh3e dev
    日立 SH-3、SH-3E ターゲット・システムです。

```

### 14.3.14 Tsquare Sparclet

開発者は、GDB を使うことによって、Sparclet ターゲット上で実行中のタスクを UNIX ホストからデバッグできるようになります。GDB は、UNIX ホスト上で実行されるコードと Sparclet ターゲット上で実行されるコードの両方を使用します。gdb は、UNIX ホスト上にインストールされて実行されます。

remotetimeout *args*

GDB はオプション remotetimeout をサポートしています。このオプションはユーザによって設定されるもので、*args* は GDB が応答を待つ秒数を表わします。

デバッグ用にコンパイルする際に、デバッグ情報を得るためには '-g' オプションを、また、ターゲット上でロードしたい位置にプログラムを再配置するためには '-Ttext' オプションを指定します。各セクションのサイズを小さくするために、'-n' オプションまたは '-N' オプションを加えるのも良いでしょう。

```
sparclet-aout-gcc prog.c -Ttext 0x12010000 -g -o prog -N
```

アドレスが意図したものと一致しているかどうかを検証するのに、objdump を使うことができます。

```
sparclet-aout-objdump --headers --syms prog
```

GDB が見つかるように UNIX の実行サーチ・パスを設定すれば、GDB を実行するための準備は完了です。UNIX ホストから gdb (インストールの方法によっては、sparclet-aout-gdb) を実行します。

GDB が起動されて、以下のプロンプトを表示します。

```
(gdb) (gdb)
```

#### 14.3.14.1 デバッグするファイルの選択

GDB の file コマンドによって、デバッグするプログラムを選択することができます。

```
(gdb) file prog
```

このコマンドを実行すると、GDB は 'prog' のシンボル・テーブルを読み込もうとします。GDB は、コマンド・サーチ・パスに含まれるディレクトリを探索することによって、そのファイルを見つけます。そのファイルがデバッグ情報付き (オプション "-g") でコンパイルされた場合は、ソース・ファイルも探します。GDB は、ディレクトリ・サーチ・パス (セクション 4.4 [ユーザ・プログラムの環境], ページ 23 参照) に含まれるディレクトリを探索することによって、そのソース・ファイルを見つけます。ファイルが見つからない場合には、次のようなメッセージを表示します。

```
prog: No such file or directory.
```

このメッセージが表示された場合には、GDB の path コマンドと dir コマンドを使って適切なディレクトリをサーチ・パスに加えてから、target コマンドを再実行します。

#### 14.3.14.2 Sparclet への接続

GDB の target コマンドによって Sparclet ターゲットに接続することができます。シリアル・ポート ttya でターゲットに接続するには、以下のように入力します。

```
(gdb) target sparclet /dev/ttya
Remote target sparclet connected to /dev/ttya
main () at ../prog.c:3
```

GDB は以下のようなメッセージを表示します。

```
Connected to ttya.
```

#### 14.3.14.3 Sparclet ダウンロード

Sparclet ターゲットへの接続が完了すると、GDB の `load` コマンドを使って、ホストからターゲットへファイルをダウンロードすることができます。ファイル名とロード・オフセットを、`load` コマンドへの引数として渡さなければなりません。ファイル形式は `aout` です。プログラムはその開始アドレスにロードされなければなりません。開始アドレスの値を知るには `objdump` を使うことができます。ロード・オフセットとは、ファイルの個々のセクションの VMA ( 仮想メモリ・アドレス ) に加算されるオフセットのことです。例えば、プログラム `'prog'` が、`text` セクションのアドレス `0x12010000`、`data` セクションのアドレス `0x12010160`、`bss` セクションのアドレス `0x12010170` にリンクされているとすると、GDB では以下のように入力します。

```
(gdb) load prog 0x12010000
Loading section .text, size 0xdb0 vma 0x12010000
```

プログラムがリンクされたアドレスとは異なるアドレスにコードがロードされた場合、どこにシンボル・テーブルをマップするかを GDB に通知するために、`section` コマンドと `add-symbol-file` コマンドを使う必要があるかもしれません。

#### 14.3.14.4 実行とデバッグ

以上により、GDB の実行制御コマンドである `b`、`step`、`run` などを使ってタスクのデバッグを開始することができます。コマンドの一覧については、GDB のマニュアルを参照してください。

```
(gdb) b main
Breakpoint 1 at 0x12010000: file prog.c, line 3.
(gdb) run
Starting program: prog
Breakpoint 1, main (argc=1, argv=0xeffff21c) at prog.c:3
3      char *symarg = 0;
(gdb) step
4      char *execarg = "hello!";
(gdb)
```

#### 14.3.15 富士通 Sparclite

`target sparclite dev`

富士通の `sparclite` ボードです。このコマンドはロードする目的でのみ使われます。プログラムをデバッグするにはさらに別のコマンドを使わなければなりません。例えば、GDB の標準リモート・プロトコルを使う `target remote dev` です。

#### 14.3.16 Tandem ST2000

GDB は、Tandem STDEBUG プロトコルを実行している Tandem ST2000 電話交換機に対して使うことができます。

ST2000 をホスト・システムに接続する方法については、製造元のマニュアルを参照してください。ST2000 が物理的に接続されれば、それをデバッグ環境として確立するには以下を実行します。

```
target st2000 dev speed
```

`dev` は通常、シリアル回線によって ST2000 と接続される `/dev/ttya` のようなシリアル装置の名前です。代わりに、`hostname:portnumber` という構文を使って (例えば、端末多重化装置経由で接続されたシリアル回線への) TCP 接続として `dev` を指定することもできます。

このターゲットに対して、`load` コマンドと `attach` コマンドは定義されていません。通常スタンドアロンで操作している場合と同様、ST2000 にユーザ・プログラムをロードしなければなりません。GDB は (シンボルのような) デバッグ用の情報を、ホスト・コンピュータ上にある同一プログラムのデバッグ・バージョンから読みとります。

ST2000 での作業を支援するために、以下の補助的な GDB コマンドが利用可能です。

`st2000 command`

`command` によって指定されるコマンドを STDEBUG モニタに送信します。利用できるコマンドについては、製造元のマニュアルを参照してください。

`connect` STDEBUG コマンド・モニタに対して制御端末を接続します。STDEBUG の操作が終了した後、`(RET)~` (Return キーに続いてチルダとピリオドを入力) または、`(RET)~(C-d)` (Return キーに続いてチルダと Control-D を入力) のいずれかを入力することによって GDB コマンド・プロンプトに戻ります。

### 14.3.17 Zilog Z8000

Zilog Z8000 をターゲットとしてデバッグするためのコンフィギュレーションが行われると、GDB には、Z8000 シミュレータが組み込まれます。

Z8000 系については、`'target sim'` によって、Z8002 (Z8000 アーキテクチャの、セグメントを持たない変種) または Z8001 (セグメントを持つ変種) をシミュレートします。シミュレータは、オブジェクト・コードを調べることで、どちらのアーキテクチャが適切であるかを認識します。

`target sim args`

シミュレートされた CPU 上でプログラムをデバッグします。シミュレータがセットアップ・オプションをサポートしている場合は、それを `args` の部分に指定します。

このターゲットを指定した後は、ホスト・コンピュータ上のプログラムをデバッグするのと同様の方法で、シミュレートされた CPU 用のプログラムをデバッグすることができます。新しいプログラムのイメージをロードするには `file` コマンドを使い、ユーザ・プログラムを実行するには `run` コマンドを使う、という具合です。

Z8000 シミュレータでは、通常のマシン・レジスタ (セクション 8.10 [レジスタ], ページ 77 参照) がすべて利用可能であるだけでなく、特別な名前を持つレジスタとして、3 つの追加情報が提供されています。

`cycles` シミュレータ内のクロック・ティックをカウントします。

`insts` シミュレータ内で実行された命令をカウントします。

`time` 1/60 秒を単位とする実行時間を示します。

これらの変数は、GDB の式の中で普通に参照することができます。例えば、`'b fputc if $cycles>5000'` は、シミュレートされたクロック・ティックが最低 5,000 回発生した後に停止するような、条件付きブレイクポイントを設定します。

## 14.4 アーキテクチャ

このセクションでは、ネイティブ・デバッグ、クロス・デバッグのいずれにおいても GDB の使用に対して影響を及ぼすような、アーキテクチャ上の特徴について説明します。

### 14.4.1 A29K

`set rstack_high_address address`

AMD 29000 ファミリ・プロセッサでは、レジスタはレジスタ・スタックと呼ばれるところに退避されます。GDB には、このスタックの大きさを知ることはできません。通常 GDB は、スタックは十分に大きいと想定します。このために、実際には存在しないメモリ位置を、GDB が参照してしまうことがあります。必要であれば、`set rstack_high_address` コマンドによってレジスタ・スタックの最終アドレスを指定することによって、この問題を回避することができます。引数はアドレスでなければなりません。‘0x’を先頭に記述することで、アドレスを 16 進数で指定することができます。

`show rstack_high_address`

AMD 29000 ファミリ・プロセッサにおけるレジスタ・スタックのカレントな上限を表示します。

### 14.4.2 Alpha

次のセクションを参照してください。

### 14.4.3 MIPS

Alpha ベースのコンピュータと MIPS ベースのコンピュータは、変わったスタック・フレームを使用しています。そのため、関数の先頭を見つけるために、GDB はときどきオブジェクト・コードを逆方向に検索する必要があります。

応答時間を改善するために（特に、このような検索を実行するのに、速度の遅いシリアル回線を使用するほかにない、組み込みアプリケーションの場合）、以下に列挙するコマンドを使用して検索量を制限するとよいでしょう。

`set heuristic-fence-post limit`

関数の先頭を検索するために GDB が検査するバイト数を、最高で *limit* バイトに制限します。*limit* に 0（デフォルト）を指定すると、無制限に検索することを意味します。*limit* に 0 以外の値を指定すると、その値が大きければ大きいほど `heuristic-fence-post` による検索バイト数も多くなり、したがって検索の実行により長い時間がかかります。

`show heuristic-fence-post`

現在の検索制限値を表示します。

これらのコマンドは、GDB が、Alpha プロセッサ上、または、MIPS プロセッサ上においてプログラムをデバッグするよう構成されている場合にのみ使用することができます。

## 15 GDB の制御

`set` コマンドによって GDB の操作方法を変更することができます。GDB によるデータの表示方法を制御するコマンドについては、セクション 8.7 [表示設定], ページ 70 を参照してください。この章では、その他の設定について説明します。

### 15.1 プロンプト

GDB は、プロンプトと呼ばれる文字列を表示することで、コマンドを受け付ける用意ができたことを示します。通常、この文字列は `(gdb)` です。`set prompt` コマンドによって、プロンプトの文字列を変更することができます。例えば、GDB を使って GDB 自体をデバッグしているときには、どちらか一方の GDB セッションのプロンプトを変更して、どちらの GDB とやりとりしているのか区別できるようにすると便利です。

注: `set prompt` は、ユーザが設定したプロンプトの後ろに空白を追加しません。ユーザは、空白で終わるプロンプト、空白で終わらないプロンプトのいずれでも設定することができます。

```
set prompt newprompt
    今後は newprompt をプロンプトとして使用するよう、GDB に指示します

show prompt
    'Gdb's prompt is: your-prompt' という形式の 1 行を表示します
```

### 15.2 コマンド編集

GDB は入力コマンドを `readline` インターフェイスによって読み取ります。この GNU ライブラリを使うことで、ユーザに対してコマンドライン・インターフェイスを提供するプログラムは、統一された振る舞いをするようになります。これを使うことの利点としては、GNU Emacs スタイルまたは `vi` スタイルによるコマンドのインライン編集、`csh` スタイルの履歴置換、複数のデバッグ・セッションにまたがるコマンド履歴の保存と呼び出しができるようになることが挙げられます。

`set` コマンドによって、GDB におけるコマンドライン編集の振る舞いを制御することができます。

```
set editing
set editing on
    コマンドライン編集を使用可能にします ( コマンドライン編集は、デフォルトの状態で使用可能です )。

set editing off
    コマンドライン編集を使用不可にします。

show editing
    コマンドライン編集が使用可能かどうかを示します。
```

### 15.3 コマンド履歴

デバッグ・セッション中にユーザが入力したコマンドを GDB に記録させることができるため、ユーザは実際に何が実行されたかを確実に知ることができます。以下のコマンドを使って、GDB のコマンド履歴機能を管理します。

```
set history filename fname
```

GDB コマンド履歴ファイルの名前を *fname* に設定します。GDB は、最初にこのファイルからコマンド履歴リストの初期値を読み取り、終了時には、このファイルにセッション中のコマンド履歴を書き込みます。コマンド履歴リストには、履歴展開機能、あるいは、後に列挙する履歴コマンド編集文字によってアクセスすることができます。このファイル名は、デフォルトでは環境変数 GDBHISTFILE の値になりますが、この変数が設定されていない場合には `./.gdb_history` (MS-DOS 上では `./_gdb_history`) になります。

```
set history save
```

```
set history save on
```

コマンド履歴をファイルの中に記録します。ファイルの名前は `set history filename` コマンドで指定可能です。デフォルトでは、このオプションは使用不可の状態になっています。

```
set history save off
```

コマンド履歴をファイルの中に記録するのを停止します。

```
set history size size
```

GDB が履歴リストの中に記録するコマンドの数を設定します。デフォルトでは、この値は環境変数 HISTSIZE の値に設定されますが、この変数が設定されていない場合は 256 になります。

履歴展開機能により、文字 `!` には特別な意味が割り当てられます。

`!` は、C 言語における論理 NOT の演算子でもあるので、履歴展開機能はデフォルトでは off になっています。 `set history expansion on` コマンドによって履歴展開を利用できるようにした場合には、( `!` を式の中で論理 NOT として使うのであれば ) `!` の後ろに空白かタブを入れることによって、それが展開されないようにする必要のある場合があります。履歴展開が有効になっている場合でも、readline の履歴機能は、`!=` や `!(` という文字列を置き換えようとはしません。

履歴展開を制御するコマンドには、以下のようなものがあります。

```
set history expansion on
```

```
set history expansion
```

履歴展開を使用可能にします。履歴展開はデフォルトでは使用不可です。

```
set history expansion off
```

履歴展開を使用不可にします。

readline のコードには、履歴編集機能や履歴展開機能に関する、より完全なドキュメントが付属しています。GNU Emacs や vi のことをよく知らない人は、このドキュメントを読むとよいでしょう。

```
show history
```

```
show history filename
```

```
show history save
```

```
show history size
```

```
show history expansion
```

これらのコマンドは、GDB の履歴パラメータの状態を表示します。単に `show history` を実行すると、4 つのパラメータの状態がすべて表示されます。



```
show commands
```

コマンド履歴中の最後の 10 個のコマンドを表示します。

```
show commands n
```

コマンド番号 *n* のコマンドを中心に、その前後の 10 個のコマンドを表示します。

```
show commands +
```

最後に表示されたコマンドに続く 10 個のコマンドを表示します。

## 15.4 画面サイズ

GDB のコマンドは、大量の情報を画面上に出力することがあります。大量の情報をすべて読むのを支援するために、GDB は 1 ページ分の情報を出力するたびに、出力を停止してユーザからの入力を求めます。出力を継続したい場合は `(RET)` キーを押し、残りの出力を破棄したい場合は `q` を入力します。また、画面幅の設定によって、どこで行を折り返すかが決まります。GDB は、単純に次の行に折り返すのではなく、出力の内容に応じて読みやすいところで折り返すよう試みます。

通常 GDB は、端末ドライバ・ソフトウェアから画面サイズを知ります。例えば UNIX 上において GDB は、termcap データベースと TERM 環境変数の値、さらに、`stty rows`、`stty cols` の設定を使います。この結果が正しくない場合、`set height` コマンドと `set width` コマンドで画面サイズの設定を変更することができます。

```
set height lpp
```

```
show height
```

```
set width cpl
```

```
show width
```

これらの `set` コマンドは、画面の高さを *lpp* 行に、幅を *cpl* 桁に指定します。対応する `show` コマンドが、現在の設定を表示します。

ゼロ行の高さを指定すると、GDB は出力がどんなに長くても、出力途中で一時停止することをしません。これは、出力先がファイルやエディタのバッファである場合に便利です。

同様に、`'set width 0'` を指定することによって、GDB に行の折り返しを行わせないようにすることもできます。

## 15.5 数値

GDB に対して 8 進、10 進、16 進の数値を慣例にしたがって入力することはいつでも可能です。8 進数は `'0'` で始まります。10 進数は `'.'` で終わります。16 進数は `'0x'` で始まります。このどれにも該当しないものは、デフォルトで 10 進数として入力されます。同様に、数値を表示するときも、特定のフォーマットが指定されていなければ、デフォルトで 10 進数として表示されます。`set radix` コマンドによって、入力、出力の両方のデフォルトを変更することができます。

```
set input-radix base
```

数値入力のデフォルトの基数を設定します。サポートされる選択肢は 10 進数の 8、10、16 です。*base* 自身はあいまいにならないように指定するか、あるいは、現在のデフォルトの基数を使用して指定します。例えば、

```
set radix 012
set radix 10.
set radix 0xa
```

は基数を 10 進数に設定します。一方、‘set radix 10’は、現在の基数を（それがどれであれ）変更しません。

```
set output-radix base
```

数値の表示に使うデフォルトの基数を設定します。サポートされる *base* の選択肢は 10 進数の 8、10、16 です。*base* 自身はあいまいにならないように指定するか、あるいは、現在のデフォルトの基数を使用して指定します。

```
show input-radix
```

数値の入力に現在使われているデフォルトの基数を表示します。

```
show output-radix
```

数値の表示に現在使われているデフォルトの基数を表示します。

## 15.6 オプションの警告およびメッセージ

デフォルトでは、GDB は内部の動作に関する情報を表示しません。性能の遅いマシンで実行している場合には、set verbose コマンドを使うとよいでしょう。これによって、GDB は、長い内部処理を実行するときにメッセージを出力することで、クラッシュと勘違いされないようにします。

現在のところ、set verbose コマンドによって制御されるメッセージは、ソース・ファイルのシンボル・テーブルを読み込み中であることを知らせるメッセージです。セクション 12.1 [ファイルを指定するコマンド]、ページ 109 の symbol-file を参照してください。

```
set verbose on
```

GDB が特定の情報メッセージを出力するようにします。

```
set verbose off
```

GDB が特定の情報メッセージを出力しないようにします。

```
show verbose
```

set verbose が on、off のどちらの状態であるかを表示します。

デフォルトでは、オブジェクト・ファイルのシンボル・テーブルに問題を検出しても、GDB はメッセージを出力しません。しかし、コンパイラをデバッグしているようなときには、このような情報があると便利かもしれません（セクション 12.2 [シンボル・ファイル読み込み時のエラー]、ページ 113 参照）。

```
set complaints limit
```

異常な型のシンボルを検出するたびに GDB が出力するメッセージの総数を *limit* 個とします。*limit* 個のメッセージを表示すると、その後は問題を検出してもメッセージを表示しないようになります。メッセージを 1 つも出力させないようにするには、*limit* にゼロを指定してください。メッセージの出力が抑止されないようにするには、*limit* に大きな値を設定してください。

```
show complaints
```

GDB が何個までシンボル異常に関するメッセージを出力できるよう設定されているかを表示します。

デフォルトでは、GDB は慎重に動作し、コマンドを本当に実行するのか確認するために、ときには馬鹿げているとさえ思えるような質問を多く尋ねてきます。例えば、既に実行中のプログラムを実行しようとする、次のように質問してきます。

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

ユーザが、このようなメッセージを見ても怯むことなく、実行したコマンドの結果を見たいのであれば、この「機能」を抑止することができます。

```
set confirm off
    確認要求を行わないようにします。

set confirm on
    確認要求を行うようにします (デフォルト)。

show confirm
    確認要求の現在の設定を表示します。
```

## 15.7 内部的な事象に関するオプションのメッセージ

```
set debug arch
    gdbarch デバッグ情報の表示を有効あるいは無効にします。デフォルトでは無効です。

show debug arch
    現在、gdbarch デバッグ情報の表示が有効になっているか無効になっているかを示します。

set debug event
    GDB イベント・デバッグ情報の表示を有効あるいは無効にします。デフォルトでは無効です。

show debug event
    現在、GDB イベント・デバッグ情報の表示が有効になっているか無効になっているかを示します。

set debug expression
    GDB 式デバッグ情報の表示を有効あるいは無効にします。デフォルトでは無効です。

show debug expression
    現在、GDB 式デバッグ情報の表示が有効になっているか無効になっているかを示します。

set debug overload
    GDB C++オーバーロード・デバッグ情報の表示を有効あるいは無効にします。これには関数のランキングなどのような情報が含まれます。デフォルトでは無効です。

show debug overload
    現在、GDB C++オーバーロード・デバッグ情報の表示が有効になっているか無効になっているかを示します。
```

`set debug remote`

シリアル回線を経由してリモート・マシンとの間で送受信したすべてのパケットに関する報告の表示を有効あるいは無効にします。情報は、GDB の標準出力ストリームに表示されます。デフォルトでは無効です。

`show debug remote`

現在、リモート・パケットの表示が有効になっているか無効になっているかを示します。

`set debug serial`

GDB のシリアル・デバッグ情報の表示を有効あるいは無効にします。デフォルトでは無効です。

`show debug serial`

現在、GDB のシリアル・デバッグ情報の表示が有効になっているか無効になっているかを示します。

`set debug target`

GDB のターゲット・デバッグ情報の表示を有効あるいは無効にします。これには、GDB のターゲット・レベルにおいて発生していることがそのまま情報として含まれています。デフォルトでは無効です。

`show debug target`

現在、GDB のターゲット・デバッグ情報の表示が有効になっているか無効になっているかを示します。

`set debug varobj`

GDB の変数オブジェクト・デバッグ情報の表示を有効あるいは無効にします。デフォルトでは無効です。

`show debug varobj`

現在、GDB の変数オブジェクト・デバッグ情報の表示が有効になっているか無効になっているかを示します。

## 16 一連のコマンドのグループ化

ブレイクポイント・コマンド ( セクション 5.1.7 [ブレイクポイント・コマンド・リスト], ページ 43 参照 ) とは別に、一連のコマンドを一括して実行するために保存する 2 つの方法を、GDB は提供しています。ユーザ定義コマンドとコマンド・ファイルがそれです。

### 16.1 ユーザ定義コマンド

ユーザ定義コマンドとは、一連の GDB コマンドに単一コマンドとしての名前を新たに割り当てたものです。これは、define コマンドによって行われます。ユーザ・コマンドは、空白で区切られた引数を最高で 10 個まで受け取ることができます。引数は、ユーザ・コマンドの中で、`$arg0...$arg9` としてアクセスすることができます。簡単な例を以下に示します。

```
define adder
  print $arg0 + $arg1 + $arg2
```

このコマンドを実行するには以下のようにします。

```
adder 1 2 3
```

上の例では、adder というコマンドを定義しています。このコマンドは、3 つの引数の合計を表示します。引数にはテキストの置換機能が働きますので、変数を参照することもできますし、複雑な式を使うこともできます。また、下位関数の呼び出しを行うこともできます。

`define commandname`

`commandname` という名前のコマンドを定義します。同じ名前のコマンドが既に存在する場合は、再定義の確認を求められます。

コマンドの定義は、define コマンドに続いて与えられる、他の GDB コマンド行から構成されます。これらのコマンドの終端は、end を含む行によって示されます。

`if`      引数として、評価の対象となる式を 1 つだけ取ります。その後一連のコマンドが続きますが、これらのコマンドは、式の評価結果が真 (ゼロ以外の値) である場合にだけ実行されます。さらに、else 行が続くことがあり、この場合は、else 行の後に、式の評価結果が偽であった場合にだけ実行される一連のコマンドが続きます。終端は、end を含む行によって示されます。

`while`    構文は if と似ています。引数として、評価の対象となる式を 1 つだけ取ります。その後は、実行されるべきコマンドが 1 行に 1 つずつ続き、最後に end がなければなりません。コマンドは、式の評価結果が真である限り、繰り返し実行されます。

`document commandname`

ユーザ定義コマンド `commandname` のドキュメントを記述します。このドキュメントは help コマンドによってアクセスできます。コマンド `commandname` は既に定義済みでなければなりません。このコマンドは、define コマンドが一連のコマンド定義を読み込むのと同様に、end で終わる一連のドキュメントを読み込みます。document コマンドの実行が完了すると、コマンド `commandname` に対して help コマンドを実行すると、ユーザの記述したドキュメントが表示されます。

document コマンドを再度実行することによって、コマンドのドキュメントを変更することができます。define コマンドによってコマンドを再定義しても、ドキュメントは変更されません。

```
help user-defined
```

すべてのユーザ定義コマンドを一覧表示します。個々のコマンドにドキュメントがあれば、その 1 行目が表示されます。

```
show user
```

```
show user commandname
```

*commandname* で指定されるコマンドを定義するのに使われた GDB コマンドを表示します (ドキュメントは表示されません)。 *commandname* を指定しないと、すべてのユーザ定義コマンドの定義が表示されます。

ユーザ定義コマンドが実行されるときに、定義内のコマンドは表示されません。定義内のコマンドがどれか 1 つでもエラーになると、ユーザ定義コマンドの実行が停止されます。

対話的に使われている場合には確認を求めてくるようなコマンドも、ユーザ定義コマンドの内部で使われている場合には確認を求めることなく処理を継続します。通常は実行中の処理に関してメッセージを表示する GDB コマンドの多くが、ユーザ定義コマンドの中から呼び出されている場合にはメッセージを表示しません。

## 16.2 ユーザ定義コマンド・フック

特別な種類のユーザ定義コマンドである、フックを定義することができます。‘hook-foo’というユーザ定義コマンドが存在すると、‘foo’というコマンドを実行するときにはいつも、‘foo’コマンドが実行される前に (引数のない) ‘hook-foo’が実行されます。

また、仮想コマンドである ‘stop’ が存在します。( ‘hook-stop’ を ) 定義すると、ユーザ・プログラムの実行が停止するたびに、その定義内のコマンドが実行されます。実行タイミングは、ブレイクポイント・コマンドの実行の直前、自動表示対象の表示の直前、および、スタック・フレームの表示の直前です。

例えば、シングル・ステップ実行をしている際には SIGALRM シグナルを無視し、通常の実行時には通常どおり処理したい場合には、以下のように定義します。

```
define hook-stop
handle SIGALRM nopass
end

define hook-run
handle SIGALRM pass
end

define hook-continue
handle SIGLARM pass
end
```

GDB のコマンドのうち、その名前が 1 つの単語から成るものには、フックを定義することができます。ただし、コマンド・エイリアスにフックを定義することはできません。フックは、コマンドの基本名に対して定義しなければなりません。例えば、bt ではなく backtrace を使います。フックの実行中にエラーが発生すると、GDB コマンドは停止します。( ユーザが実際に入力したコマンドが実行する機会を与えられる前に ) GDB はプロンプトを表示します。

既知のコマンドのいずれにも対応しないフックを定義しようとすると、define コマンドは警告メッセージを表示します。

### 16.3 コマンド・ファイル

GDB のコマンド・ファイルとは、各行が GDB コマンドとなっているファイルのことです。( 行の先頭が # の ) コメントも含めることができます。コマンド・ファイル内の空行は何も実行しません。それは、端末上での実行の場合とは異なり、最後に実行されたコマンドの繰り返しを意味しません。

GDB を起動すると、自動的に初期化ファイルからコマンドを読み込んで実行します。これは、UNIX 上では `‘.gdbinit’` という名前のファイルであり、DOS/Windows 上では `‘gdb.ini’` という名前のファイルです。GDB は、ユーザのホーム・ディレクトリ<sup>1</sup> に初期化ファイルがあればまずそれを読み込み、続いてコマンドライン・オプションとオペランドを処理した後、カレントな作業ディレクトリに初期化ファイルがあればそれを読み込みます。このように動くのは、ユーザのホーム・ディレクトリに初期化ファイルを置くことで、コマンドライン上のオプションやオペランドの処理に影響を与える ( `set complaints` のような ) オプションを設定することができるようにするためです。 `‘-nx’` オプションを使用すると、初期化ファイルは実行されません。セクション 2.1.2 [モードの選択], ページ 11 参照。

GDB のいくつかの構成では、初期化ファイルは異なる名前で行われています ( このような環境では、特別な形式の GDB が他の形式の GDB と共存する必要があり、そのために特別なバージョンの GDB の初期化ファイルには異なる名前が付けられます )。特別な名前の初期化ファイルを持つ環境には、以下のようなものがあります。

- VxWorks ( Wind River Systems 社のリアルタイム OS ) : `‘.vxgdbinit’`
- OS68K ( Enea Data Systems 社のリアルタイム OS ) : `‘.os68gdbinit’`
- ES-1800 ( Ericsson Telecom 社の AB M68000 エミュレータ ) : `‘.esgdbinit’`

また、`source` コマンドによって、コマンド・ファイルの実行を要求することもできます。

`source filename`

コマンド・ファイル `filename` を実行します。

コマンド・ファイルの各行は順番に実行されます。コマンドの実行時に、そのコマンドは表示されません。どれか 1 つでもコマンドがエラーになると、コマンド・ファイルの実行は停止されます。対話的に使われている場合には確認を求めてくるようなコマンドも、コマンド・ファイル内で使われている場合は確認を求めることなく処理を続けます。通常は実行中の処理についてメッセージを表示する GDB コマンドの多くが、コマンド・ファイルの中から呼び出されている場合にはメッセージを表示しません。

### 16.4 制御された出力を得るためのコマンド

コマンド・ファイルやユーザ定義コマンドの実行中には、通常の GDB の出力は抑止されます。唯一出力されるのは、定義内のコマンドが明示的に表示するメッセージだけです。ここでは、ユーザが希望するとおりの出力を生成するのに役に立つ、3 つのコマンドについて説明します。

`echo text` `text` を表示します。通常は表示されない文字も、C のエスケープ・シーケンスを使うことで `text` の中に含めることができます。例えば、改行コードを表示するには `‘\n’` を使います。明示的に指定しない限り、改行コードは表示されません。標準的

---

<sup>1</sup> 原注 : DOS/Windows システムでは、HOME 環境変数の指すディレクトリがホーム・ディレクトリです。

な C のエスケープ・シーケンスに加えて、バックスラッシュの後ろに空白を置くことで、空白が表わされます。これは、先頭や末尾に空白のある文字列を表示するのに便利です。というのは、こうしないと、すべての引数の先頭や末尾の空白は削除されるからです。‘ and foo = ’を表示するには、‘echo \ and foo = \’を実行してください。

C と同様、*text* の末尾にバックスラッシュを置くことで、コマンドを次の行以降に継続することができます。例えば、

```
echo This is some text\n\
which is continued\n\
onto several lines.\n
```

は

```
echo This is some text\n
echo which is continued\n
echo onto several lines.\n
```

と同じ出力をもたらします。

*output expression*

*expression* の値を表示し、それ以外には何も表示しません。改行コードも、‘\$nn = ’も表示されません。*expression* の値は値履歴には入りません。式の詳細については、セクション 8.1 [式], ページ 63 を参照してください。

*output/fmt expression*

*expression* の値を、*fmt* で指定されるフォーマットで表示します。print コマンドと同じフォーマットを指定することができます。詳細については、セクション 8.4 [出力フォーマット], ページ 66 を参照してください。

*printf string, expressions...*

*string* で指定された文字列にしたがって *expressions* の値を表示します。複数の *expressions* はカンマで区切られ、数値かポインタのいずれかを指定できます。これらの値は、ユーザ・プログラムから C のサブルーチン

```
printf (string, expressions...);
```

を実行した場合と同様に、*string* の指定にしたがって表示されます。

例えば、次のようにして 2 つの値を 16 進数で表示することができます。

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

フォーマットを指定する文字列の中で使えるバックスラッシュ・エスケープ・シーケンスは、バックスラッシュとそれに続く単一文字から構成される簡単なものだけです。



## 17 GNU Emacs の中での GDB の使い方

GDB でデバッグ中のプログラムのソース・ファイルを GNU Emacs を使って参照（および編集）するための、特別なインターフェイスが提供されています。

このインターフェイスを使うには、Emacs の中で `M-x gdb` コマンドを使います。デバッグしたい実行ファイルを引数として指定してください。このコマンドは、GDB を Emacs のサブプロセスとして起動し、新しく作成した Emacs バッファを通じて入出力を行います。

Emacs の中での GDB の使い方は、通常の GDB の使い方とはほぼ同様ですが、2 つ相違点があります。

- 「端末」へのすべての入出力は Emacs バッファへ送られる。

これは、GDB コマンドとその出力、および、デバッグ対象のユーザ・プログラムによる入出力の両方に適用されます。

以前に実行したコマンド・テキストをコピーして再入力することができるので便利です。出力された部分に関しても同様のことができます。

Emacs の Shell モードで利用可能なすべての機能を、ユーザ・プログラムとのやりとりで使うことができます。特に、通常どおりにシグナルを送信することができます。例えば、`C-c C-c` で割り込みシグナルを、`C-c C-z` でストップ・シグナルを発生させることができます。

- GDB は Emacs を使ってソース・コードを表示する。

GDB がスタック・フレームを表示するときにはいつでも、Emacs がそのフレームのソース・ファイルを自動的に見つけて、カレント行の左側の余白に矢印（`=>`）を表示します。Emacs はソース・コードを別バッファに表示し、スクリーンを 2 つに分けて、GDB セッションとソースをもに表示します。

GDB の `list` コマンドや `search` コマンドを明示的に使えば、通常どおりの出力を生成することもできますが、これらを Emacs から使う理由はおそらくないでしょう。

注意：ユーザ・プログラムの存在するディレクトリがユーザのカレント・ディレクトリでない場合、ソース・ファイルの存在場所について Emacs は簡単に混乱に陥ります。このような場合、ソースを表示するための追加のディスプレイ・バッファは表示されません。GDB は、ユーザの環境変数 `PATH` のリストの中にあるディレクトリを探索してプログラムを見つけ出しますので、GDB の入出力セッションは通常どおり進行します。しかし Emacs は、このような状況においてソース・ファイルを見つけ出すのに十分な情報を GDB から受け取っていません。この問題を回避するには、ユーザ・プログラムの存在するディレクトリから GDB モードを開始するか、`M-x gdb` の引数の入力を求められたときに、絶対パスでファイル名を指定します。

Emacs の既存の GDB バッファから、デバッグ対象をどこかほかの場所にあるプログラムに変更する目的で GDB の `file` コマンドを使うと、同様の混乱の発生することがあります。

デフォルトでは、`M-x gdb` は `'gdb'` という名前のプログラムを呼び出します。別の名前で GDB を呼び出す必要がある場合（例えば、異なる構成の GDB を別の名前で持っているような場合）は、Emacs の `gdb-command-name` という変数を設定します。例えば

```
(setq gdb-command-name "mygdb")
```

（を `M-:` または `ESC :` に続けて入力するか、あるいは、`*scratch*` バッファまたは `' .emacs '` ファイルに入力することで）Emacs は `gdb` の代わりに「`mygdb`」という名前のプログラムを呼び出します。

GDB の I/O バッファでは、標準的な Shell モードのコマンドに加えて、以下のような特別な Emacs コマンドを使うことができます。

|                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>C-h m</code>         | Emacs の GDB モードの機能に関する説明を表示します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>M-s</code>           | GDB の <code>step</code> コマンドのように、ソース行を 1 行実行します。さらに、カレントなファイルとそこにおける位置を示すために、表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>M-n</code>           | GDB の <code>next</code> コマンドのように、関数呼び出しをすべてスキップして、現在の関数内の次のソース行まで実行を進めます。さらに、カレントなファイルとそこにおける位置を示すために、表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>M-i</code>           | GDB の <code>stepi</code> コマンドのように、1 命令を実行します。必要に応じて表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>M-x gdb-nexti</code> | GDB の <code>nexti</code> コマンドを使って、次の命令まで実行します。必要に応じて表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>C-c C-f</code>       | GDB の <code>finish</code> コマンドのように、選択されたスタック・フレームを終了するまで実行を継続します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>M-c</code>           | GDB の <code>continue</code> コマンドのように、ユーザ・プログラムの実行を継続します。<br>注意：Emacs v19 では、このコマンドは <code>C-c C-p</code> です。                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>M-u</code>           | GDB の <code>up</code> コマンドのように、数値引数によって示される数だけ上位のフレームに移動します（セクション “Numeric Arguments” in <i>The GNU Emacs Manual</i> <sup>1</sup> を参照してください）。<br>注意：Emacs v19 では、このコマンドは <code>C-c C-u</code> です。                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>M-d</code>           | GDB の <code>down</code> コマンドのように、数値引数によって示される数だけ下位のフレームに移動します。<br>注意：Emacs v19 では、このコマンドは <code>C-c C-d</code> です。                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>C-x &amp;</code>     | カーソルの位置にある数値を読み取り、GDB の I/O バッファの末尾に挿入します。例えば、以前に表示されたアドレスの前後のコードを逆アセンブルしたいとしましょう。この場合、まず <code>disassemble</code> と入力し、次に表示されたアドレスのところにカーソルを移動し、 <code>disassemble</code> への引数を <code>C-x &amp;</code> で読み取ります。<br><br>gdb-print-command リストの要素を定義することによって、これをさらにカスタマイズすることができます。これが定義されていると、 <code>C-x &amp;</code> で入手した数値が挿入される前に、それをフォーマットしたり、処理したりすることができるようになります。 <code>C-x &amp;</code> に数値引数を指定すると、特別なフォーマット処理を必要としているという意味になり、その数値がリストの要素を取得するためのインデックスとなります。リストの要素が文字列の場合は、挿入される数値は Emacs の <code>format</code> 関数によってフォーマットされます。リストの要素が文字列以外の場合は、その数値が、対応するリスト要素への引数として渡されます。 |

どのソース・ファイルが表示されている場合でも、Emacs の `C-x SPC` (`gdb-break`) コマンドは、ポイントの置かれているソース行にブレイクポイントをセットするよう GDB に対して指示します。

<sup>1</sup> 訳注：GNU Emacs 19 マニュアル（星雲社）の「ニューメリック引数」、GNU Emacs マニュアル（共立出版）の「数引数」に、日本語訳があります。

ソースを表示中のバッファを誤って削除してしまった場合に、それを再表示させる簡単な方法は、GDB バッファの中で `f` コマンドを入力して、フレーム表示を要求することです。Emacs 配下では、カレント・フレームのコンテキストを表示するために必要であれば、ソース・バッファが再作成されます。

Emacs によって表示されるソース・ファイルは、通常どおりの方法でソース・ファイルにアクセスする、普通の Emacs バッファによって表示されます。そうしたいのであれば、これらのバッファの中でファイルの編集を行うこともできますが、GDB と Emacs の間で行番号に関する情報が交換されていることを頭に入れておいてください。テキストに行を挿入したり、削除したりすると、GDB の認識しているソース行番号は、実際のコードと正しく対応しなくなってしまうます。



## 18 GDB 註釈

この章では、GDB における註釈機能について説明します。註釈は、グラフィカル・ユーザ・インターフェイス、あるいは、比較的上位のレベルにおいて GDB とやりとりを行う他の同様のプログラムと GDB との間を仲立ちすることを目的として設計されています。

### 18.1 註釈

註釈を生成するには、`--annotate=2` オプションを指定して GDB を起動します。

註釈は改行文字で始まり、その後ろに、2 つの `'control-z'` と註釈名が続きます。註釈に関連付けられた追加情報がない場合には、註釈名の直後に改行が続きます。追加情報がある場合は、註釈名の後ろに空白を 1 つ空けて追加情報が続き、最後に改行がきます。追加情報の中に改行文字を含めることはできません。

改行文字と 2 つの `'control-z'` で始まらない出力はすべて、GDB の出力を文字どおりに表わしています。現在のところ、GDB が改行文字に続けて 2 つの `'control-z'` 文字を出力する必要性はないのですが、そのような必要性が出てくれば、これらの 3 文字の出力を意味する `'escape'` 註釈によって註釈を拡張することができます。

註釈付きで GDB を起動する単純な例を以下に示します。

```
$ 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
$
```

ここで `'quit'` は、GDB への入力です。それ以外は GDB からの出力です。`'^Z^Z'` (`'^Z'` は `'control-z'` 文字を表わします) で始まる 3 行が註釈です。それ以外は GDB からの出力です。

### 18.2 server 接頭語

ユーザによって認識されている状態に影響を与えることなく GDB に対するコマンドを発行するためには、そのコマンドの前に `'server'` という文字列を付けてください。これは、このコマンドがコマンド履歴に影響を与えないということを意味しています。すなわち、行内にほかの文字がない状態で `(RET)` が押された場合にどのコマンドを繰り返すべきかという点に関する GDB の認識に影響を与えないということです。

server 接頭語は、値履歴への値の記録にも影響を与えません。ある値を、値履歴に記録することなく表示させたい場合には、`print`コマンドの代わりに `output` コマンドを使ってください。

### 18.3 値

様々なコンテキストにおいて値を表示する際に、GDB は註釈を使って、その値をまわりのテキストから区切ります。

ある値が、`print` コマンドを使って表示され、値ヒストリに追加されると、註釈は以下のようになります。

```
^Z^Zvalue-history-begin history-number value-flags
history-string
^Z^Zvalue-history-value
the-value
^Z^Zvalue-history-end
```

*history-number* は、その値の値履歴の中における番号です。*history-string* は、`'$5 = '` のような文字列で、その後に値が表示されることをユーザに示すものです。*the-value* は、値そのものに対応する出力です。*value-flags* は、間接参照できる値については `'*`、間接参照できない値については `'-` です。

値が値履歴に追加されない場合 ( 不正な浮動小数点数である場合や、`output` コマンドを使って表示される場合 ) の註釈も同様です。

```
^Z^Zvalue-begin value-flags
the-value
^Z^Zvalue-end
```

GDB が ( 例えば、`backtrace` コマンドによる出力の中で ) 関数の引数を表示する際の註釈は以下のようになります。

```
^Z^Zarg-begin
argument-name
^Z^Zarg-name-end
separator-string
^Z^Zarg-value value-flags
the-value
^Z^Zarg-end
```

*argument-name* は引数名です。*separator-string* は、ユーザの利便のために、その名前と値を区切る ( 例えば `'= '` のような ) テキストです。*value-flags* と *the-value* の意味は、`value-history-begin` の註釈の場合と同一です。

構造体を表示する際には、GDB は以下のようにな註釈します。

```
^Z^Zfield-begin value-flags
field-name
^Z^Zfield-name-end
separator-string
^Z^Zfield-value
the-value
^Z^Zfield-end
```

*field-name* はフィールド名です。*separator-string* は、ユーザの利便のために、その名前と値を区切る ( 例えば `'= '` のような ) テキストです。*value-flags* と *the-value* の意味は、`value-history-begin` の註釈の場合と同一です。

配列を表示する際には、GDB は以下のように註釈します。

```
^Z^Zarray-section-begin array-index value-flags
```

*array-index* は、註釈される最初の要素のインデックスです。*value-flags* の意味は、*value-history-begin* の註釈の場合と同一です。この後に、任意の数の要素が続きます。要素は以下のように単一要素であることもあります。

```
‘,’ whitespace          ; 最初の要素については省略
the-value
^Z^Zelt
```

また、繰り返し要素であることもあります。

```
‘,’ whitespace          ; 最初の要素については省略
the-value
^Z^Zelt-rep number-of-repitions
repetition-string
^Z^Zelt-rep-end
```

いずれの場合でも、*the-value* は要素の値を出力したものであり、*whitespace* には、空白、タブ、改行が含まれます。繰り返し要素の場合、*number-of-repitions* は、その値を持つ配列要素が何個連続しているかを示す数です。また、*repetition-string* は、繰り返しが表現されていることをユーザに知らせるよう設計された文字列です。

配列のすべての要素が出力されると、配列の註釈は以下によって終了します。

```
^Z^Zarray-section-end
```

## 18.4 フレーム

GDB がフレームを表示する際には、常に註釈が加えられます。例を挙げると、GDB が停止したときに表示されるフレームや、*backtrace* コマンドや *up* コマンド等の出力がこれに該当します。フレームに対する註釈の先頭は以下ようになります。

```
^Z^Zframe-begin level address
level-string
```

*level* はフレームの番号です（最下位のフレームが 0 で、それ以外のフレームは正の数を取ります）。*address* は、そのフレームにおいて実行中のコードのアドレスです。*level-string* は、ユーザにレベル（フレーム番号）を知らせるよう設計された文字列です。*address* は、‘0x’の後ろに小文字の 16 進数が 1 つ以上続く形式になります。（これが言語に依存しないことに注意してください）。フレームの末尾は以下ようになります。

```
^Z^Zframe-end
```

これらの註釈の間にフレームの本体が入ります。これは、以下のものから構成されます。

- 

```
^Z^Zfunction-call
function-call-string
```

*function-call-string* は、デバッグ対象のプログラム中の関数に対する GDB からの関数呼び出しに対してこのフレームが関連付けられていることを、ユーザに知らせるよう設計されたテキストです。

- 

```
^Z^Zsignal-handler-caller
signal-handler-caller-string
```

*signal-handler-caller-string* は、オペレーティング・システムによるシグナル・ハンドラ呼び出し機構に対してこのフレームが関連付けられていることを、ユーザに知らせよう設計されたテキストです（これは、シグナル・ハンドラそのものに対応したフレームではなく、シグナル・ハンドラを呼び出したフレームです）。

- 通常のフレーム。

これは（ユーザが興味をもって観察するような情報であると考えられるかどうかには依存しますが）以下のような情報で始まる場合があります。

```
^Z^Zframe-address
address
^Z^Zframe-address-end
separator-string
```

*address* は、そのフレームにおいて実行中のアドレスです（*frame-begin* 註釈におけるものと同一のアドレスですが、ユーザが利用することを想定した形式で表示されます。具体的には、構文が言語に依存して変わります）。*separator-string* は、ユーザの利便のために、このアドレスと後続の情報を区切るための文字列です。

続いて以下の情報が表示されます。

```
^Z^Zframe-function-name
function-name
^Z^Zframe-args
arguments
```

*function-name* は、そのフレームにおいて実行中の関数名です。関数名が不明の場合は ‘??’ となります。*arguments* はフレームに対する引数でまわりが丸括弧 ( ) で囲まれます（個々の引数に対しても個別に註釈が加えられます。セクション 18.3 [値], ページ 172 参照）。

ソースの情報が利用できる場合、それに対する参照情報が次に表示されます。

```
^Z^Zframe-source-begin
source-intro-string
^Z^Zframe-source-file
filename
^Z^Zframe-source-file-end
:
^Z^Zframe-source-line
line-number
^Z^Zframe-source-end
```

*source-intro-string* は、ユーザの利便のために、この参照情報をそれ以前に表示されたテキストから区切るものです。*filename* はソース・ファイル名、*line-number* はそのファイル中における行番号です（最初の行が行 1 となります）。

GDB がフレームの発生源（ライブラリ名、ロード・セグメント名等。現在のところこれは RS/6000 上でのみ実行されます）に関する情報を表示する際の註釈は以下のようになります。

```
^Z^Zframe-where
information
```

続いて、このフレームに対応するソースが実際に表示されるのであれば（例えば *backtrace* コマンドの出力ではこのようなことは行われません）、*source* に対する註釈（セクション 18.11 [ソースの表示], ページ 178 参照）が表示されます。この註釈は、ほとんどの註釈とは異なり、通常の場合に出力されるテキストに追加して表示されるのではなく、通常の場合に出力されるテキストの代わりに表示されます。



## 18.5 表示

GDB が `display` コマンドによって何かを表示するよう通知された場合、表示結果に註釈が加えられます。

```

^Z^Zdisplay-begin
number
^Z^Zdisplay-number-end
number-separator
^Z^Zdisplay-format
format
^Z^Zdisplay-expression
expression
^Z^Zdisplay-expression-end
expression-separator
^Z^Zdisplay-value
value
^Z^Zdisplay-end

```

`number` は表示の番号です。`number-separator` は、ユーザのために、この番号を後続の情報と区切るためのものです。`format` には、サイズ、形式、および、値の表示方法に関するその他の情報が含まれます。`expression` は、表示対象となった式です。`expression-separator` は、ユーザのために、この式を後続のテキストと区切るためのものです。`value` が、実際に表示された値です。

## 18.6 GDB の入力に対する註釈

プロンプトを表示して入力を要求する際、GDB はそのことを註釈で示すことによって、いつ情報を送信すべきなのか、また、あるコマンドからの出力の終わりがどこなのかを判別できるようにします。

異なる種類の入力はそれぞれ異なる入力タイプを持ちます。個々の入力タイプには 3 種類の註釈があります。`pre`-註釈は、出力されているプロンプトの先頭を示すものです。修飾されていない註釈<sup>1</sup> はプロンプトの終端を示します。`post`-註釈は、入力に関連付けられているかもしれないエコー出力の終端を示します。例えば、`prompt` 入力タイプは以下のような註釈を提供します。

```

^Z^Zpre-prompt
^Z^Zprompt
^Z^Zpost-prompt

```

入力タイプには以下のものがあります。

|                              |                                                                                                  |
|------------------------------|--------------------------------------------------------------------------------------------------|
| <code>prompt</code>          | GDB がコマンドの入力を要求しているときに出力されます。( GDB のメイン・プロンプトです )                                                |
| <code>commands</code>        | GDB が、例えば <code>commands</code> コマンドにおけるように、コマンドの集合を要求しているときに出力されます。入力される個々のコマンドに対して、註釈が繰り返されます。 |
| <code>overload-choice</code> | いくつかのオーバーロードされた関数群の中から選択するよう、GDB がユーザに対して要求しているときに出力されます。                                        |

<sup>1</sup> 訳注：`pre`-や `post`-の付かない註釈

`query` 潜在的な危険性を持つ操作を実行してもかまわないことを確認するよう、GDB がユーザに対して要求しているときに出力されます。

`prompt-for-continue`

処理を継続するためにリターン・キーを押すよう、GDB がユーザに対して要求しているときに出力されます。注：これが正しく機能すると期待しないでください。代わりに、`set height 0`を使ってプロンプトを抑止してください。註釈がある場合、行数カウントに不具合があるからです。

## 18.7 エラー

`^Z^Zquit`

この註釈は、GDB が割り込みに反応する直前に出力されます。

`^Z^Zerror`

この註釈は、GDB がエラーに反応する直前に出力されます。

`quit` 註釈と `error` 註釈は、GDB による未完了の註釈が突然終了してしまったことを示唆しています。例えば、`value-history-begin`註釈の後に `error`が出力された場合、対応する `value-history-end`が表示されることを期待することはできません。もっとも、それが表示されないことを期待することもできません。`error` 註釈は、GDB が即座にトップ・レベルにまで一気に復帰することを必ずしも意味するわけではありません。

`quit` 註釈や `error` 註釈に先行して、以下の表示が行われることがあります。

`^Z^Zerror-begin`

これと `quit` 註釈や `error` 註釈との間の出力はすべてエラー・メッセージです。

今のところ警告メッセージにはまだ註釈が加えられていません。

## 18.8 ブ레이크ポイントに関する情報

`info breakpoints` コマンドの出力に対しては、以下のような註釈が加えられます。

`^Z^Zbreakpoints-headers`

*header-entry*

`^Z^Zbreakpoints-table`

*header-entry* の構文はエントリ (下記参照) のそれと同一ですが、データの代わりに、個々のフィールドの意味をユーザに知らせるための文字列を含みます。これに続いて任意個数のエントリが続きます。あるフィールドがこのエントリには適合しない場合、それは省略されます。フィールドの末尾には空白類が含まれることがあります。個々のエントリは、以下のものから構成されます。

`^Z^Zrecord`

`^Z^Zfield 0`

*number*

`^Z^Zfield 1`

*type*

`^Z^Zfield 2`

*disposition*

`^Z^Zfield 3`

*enable*

`^Z^Zfield 4`

```

address
^Z^Zfield 5
what
^Z^Zfield 6
frame
^Z^Zfield 7
condition
^Z^Zfield 8
ignore-count
^Z^Zfield 9
commands

```

*address* はユーザが利用することを想定したものである点に注意してください。つまり、その構文は言語に依存して変わるということです。

出力の終端は以下のようになります。

```
^Z^Zbreakpoints-table-end
```

## 18.9 失効通知

以下の註釈は、何らかの状態が変更された可能性があることを通知するものです。

```
^Z^Zframes-invalid
```

フレーム（例えば `backtrace` コマンドからの出力）が変更された可能性があることを示します。

```
^Z^Zbreakpoints-invalid
```

ブレイクポイントが変更された可能性があることを示します。例えば、ユーザがブレイクポイントを追加もしくは削除した場合です。

## 18.10 プログラムの実行

`step` や `continue` のような GDB コマンドによってプログラムの実行が開始されると

```
^Z^Zstarting
```

が出力されます。プログラムが停止すると

```
^Z^Zstopped
```

が出力されます。`stopped` 註釈に先行する様々な註釈によって、どのようにプログラムが停止したのかが説明されます。

```
^Z^Zexited exit-status
```

プログラムが終了（`exit`）したときに表示されます。*exit-status* は終了ステータス（正常終了の場合は 0、それ以外の場合は 0 以外の値）です。

```
^Z^Zsignalled
```

プログラムがシグナルを受信したために終了したときに表示されます。`^Z^Zsignalled` の後に以下の註釈が続きます。

```

intro-text
^Z^Zsignal-name
name

```

```

^Z^Zsignal-name-end
middle-text
^Z^Zsignal-string
string
^Z^Zsignal-string-end
end-text

```

*name* は、SIGILLやSIGSEGVのようなシグナルの名前です。*string* はシグナルの説明です。例えば、Illegal InstructionやSegmentation faultのような文字列です。*intro-text*、*middle-text*、*end-text* はユーザのために表示されるもので、特定の形式はありません。

^Z^Zsignal

この註釈の構文は *signalled* のそれによく似ていますが、これによって GDB が言っていることは、プログラムがシグナルを受信したということだけです。シグナルを受信したために停止したとは言っていない。

^Z^Zbreakpoint *number*

プログラムが番号 *number* のブレイクポイントに到達したときに表示されます。

^Z^Zwatchpoint *number*

プログラムが番号 *number* のウォッチポイントに到達したときに表示されます。

## 18.11 ソースの表示

ソース・コードを表示する代わりに、以下のような註釈が使われます。

^Z^Zsource *filename:line:character:middle:addr*

*filename* はソース・ファイルの絶対ファイル名です。*line* は、そのファイルの中の行番号です（ファイルの中の先頭行は 1 となります）。*character* は、そのファイルの中の文字位置です（ファイルの中の先頭文字は 0 となります）。（これは、ほとんどのデバッグ形式において、必ず行の先頭を指すこととなります）。*middle* の部分は、*addr* が行の途中を指すときは 'middle' となり、*addr* が行の先頭を指すときは 'beg' となります。*addr* は、表示されているソースに関連付けられているターゲット・プログラム内のアドレスです。*addr* は、'0x' の後ろに小文字の 16 進数が 1 つ以上続く形式となります（これは言語に依存しないことに注意してください）。

## 18.12 将来追加する可能性のある註釈

- target-invalid

ターゲットが変更された可能性があることを示す註釈

（レジスタ、ヒープ内容、実行ステータスなど）

性能を向上するため、より高い精度をもって 'registers-invalid' や

'all-registers-invalid' を検出できるように徐々にしていきたいと考えています。

- set/show パラメータに対する（失効通知を含む）体系的な註釈

- 失効通知の候補リストを返す 'info'

## 19 GDB/MI インターフェイス

### 機能と目的

GDB/MI は、GDB に対する行ベース・マシン用のテキスト・インターフェイスです。これは、より大きなシステムの中における小さなコンポーネントとしてこのデバッガを使うシステムの開発をサポートすることを、特に意識しています。

この章は、GDB/MI インターフェイスの仕様書であり、リファレンス・マニュアルの形式で記述されています。

GDB/MI は現在まだ開発中のものですので、以下に説明されている機能のいくつかは不完全であったり変更されたりする可能性があります。注意してください。

### 表記法と用語

この章では、以下の表記を使います。

- `|` は 2 つの選択枝を区切ります。
- `[ something ]` は、*something* が任意選択である（必須ではない）ことを示します。それはあってもなくても構いません。
- `( group )*` は、丸括弧 `()` の中の *group* が 0 回以上繰り返されることを示します。
- `( group )+` は、丸括弧 `()` の中の *group* が 1 回以上繰り返されることを示します。
- `"string"` は字義どおりの *string* を意味します。

### 謝辞

アルファベット順 : Andrew Cagney、Fernando Nasser、Stan Shebs、Elena Zannoni

## 19.1 GDB/MI コマンド構文

### 19.1.1 GDB/MI 入力構文

*command*  $\mapsto$

*cli-command* | *mi-command*

*cli-command*  $\mapsto$

`[ token ] cli-command nl`

*cli-command* は既存の任意の GDB CLI コマンド。

*mi-command*  $\mapsto$

`[ token ] "-" operation ( " " option )* [ " --" ] ( " " parameter )* nl`

*token*  $\mapsto$  "連続した任意の数字"

*option*  $\mapsto$  `"-" parameter [ " " parameter ]`

*parameter*  $\mapsto$

*operation*  $\mapsto$

このドキュメントにおいて説明される任意のオペレーション

*non-blank-sequence*  $\mapsto$

"-", *nl*、""、" "のような特殊な文字を含まないという条件を満足する任意の文字列

*c-string*  $\mapsto$

"" 7ビットの ISO C 文字列 ""

*nl*  $\mapsto$

CR | CR-LF

注:

- CLI コマンドは現在のところはまだ、MI インタープリタによって処理されます。その出力については後述します。
- *token*が存在すると、コマンドの終了時にそれが返されます。
- MI コマンドの中には、パラメータ・リストの一部としてオプション引数を受け付けるものがあります。個々のオプションは先頭の '-' (ダッシュ)によって識別され、その後ろにオプション引数パラメータが続くことがあります。オプションはパラメータ・リストの先頭に現れ、通常のパラメータとの境界は '--'を使って指定することができます(これは、パラメータの中にダッシュで始まるものがあるときに便利です)。

プラグマティクス:

- (デバッグのための)既存の CLI 構文に容易にアクセスできるようにしたいと考えています。
- MI オペレーションを簡単に見つけることができるようにしたいと考えています。

### 19.1.2 GDB/MI 出力構文

GDB/MI からの出力は、0 個以上の帯域外レコード<sup>1</sup>から構成されます。この後に結果レコードが 1 つ続くことがあります。この結果レコードは最後に実行されたコマンドによるものです。連続した出力レコードは '(gdb)'により終了します。

入力コマンドの接頭語として *token*が付いたのであれば、そのコマンドに対応する出力にも同一の *token*が接頭語として付くことになります。

*output*  $\mapsto$  ( *out-of-band-record* ) \* [ *result-record* ] "(gdb)" *nl*

*result-record*  $\mapsto$

[ *token* ] "^" *result-class* ( "," *result* ) \* *nl*

*out-of-band-record*  $\mapsto$

*async-record* | *stream-record*

*async-record*  $\mapsto$

*exec-async-output* | *status-async-output* | *notify-async-output*

*exec-async-output*  $\mapsto$

[ *token* ] "\*" *async-output*

*status-async-output*  $\mapsto$

[ *token* ] "+" *async-output*

---

<sup>1</sup> 訳注: セクション 19.3.3 [GDB/MI 帯域外レコード], ページ 184 参照

```

notify-async-output ↦
    [ token ] "=" async-output

async-output ↦
    async-class ( "," result ) * nl

result-class ↦
    "done" | "running" | "connected" | "error" | "exit"

async-class ↦
    "stopped" ( 必要に応じて追加していく予定ですが、まだ開発中です )

result ↦    [ string "=" ] value

value ↦    const | "{ " result ( "," result ) * " }"

const ↦    c-string

stream-record ↦
    console-stream-output | target-stream-output | log-stream-output

console-stream-output ↦
    "~" c-string

target-stream-output ↦
    "@" c-string

log-stream-output ↦
    "&" c-string

nl ↦        CR | CR-LF

token ↦     連続した任意の数字

```

さらに、以下のものが現在開発中です。

*query*        このアクションは現在未定義です。

注:

- 連続した出力はすべて、ピリオドを含む単一行によって終了します。
- *token* は、対応するリクエスト( 要求 )から取られます。実行コマンドが ‘-exec-interrupt’ コマンドによって割り込まれる場合、‘\*stopped’ メッセージに関連付けられる *token* は、割り込みコマンドの *token* ではなく、もともとの実行コマンドの *token* です。
- *status-async-output* には、時間のかかるオペレーションの進行具合に関する現在のステータス情報が含まれます。これは破棄することもできます。すべてのステータス出力には、接頭語として ‘+’ が付きます。
- *exec-async-output* には、ターゲットの非同期的な状態変化( 実行停止、実行開始、消滅 )に関する情報が含まれます。これらすべての非同期出力には、接頭語として ‘\*’ が付きます。
- *notify-async-output* には、クライアントが対処すべき補足情報( 例えば、新たなブレイクポイント情報 )が含まれます。すべての通知出力には、接頭語として ‘=’ が付きます。
- *console-stream-output* は、コンソールにおいてそのまま表示されるべき出力です。それは、CLI コマンドに対するテキストによる応答です。すべてのコンソール出力には、接頭語として ‘~’ が付きます。

- *target-stream-output* は、ターゲット・プログラムにより生成される出力です。すべてのターゲット出力には、接頭語として '@' が付きます。
- *log-stream-output* は、GDB 内部からの出力テキストです。例えば、エラー・ログの一部として表示されるべきメッセージです。すべてのログ出力には、接頭語として '&' が付きます。

さまざまな出力レコードに関する詳細については、セクション 19.3.2 [GDB/MI ストリーム・レコード], ページ 183 を参照してください。

現在の出力構文に対して提案されている改訂内容については、セクション 19.15 [GDB/MI 出力構文の変更案], ページ 233 を参照してください。

### 19.1.3 GDB/MI とのやりとりの簡単な例

このサブセクションでは、GDB/MI インターフェイスを使ったやりとりの簡単な例をいくつか紹介します。これらの例では、'->' が GDB/MI に入力として渡される行を、'<-' が GDB/MI からの出力をそれぞれ示しています。

#### ターゲットの停止

以下に、下位プロセスを停止する例を示します。

```
-> -stop
<- (gdb)
```

この後に以下が表示されます。

```
<- *stop,reason="stop",address="0x123",source="a.c:123"
<- (gdb)
```

#### 簡単な CLI コマンド

以下に、GDB/MI を経由して CLI に渡される簡単な CLI コマンドの例を示します。

```
-> print 1+2
<- ~3\n
<- (gdb)
```

#### 副作用を持つコマンド

```
-> -symbol-file xyz.exe
<- *breakpoint,nr="3",address="0x123",source="a.c:123"
<- (gdb)
```

#### 不正なコマンド

以下に、存在しないコマンドを渡した場合に何が起こるかを示します。

```
-> -rubbish
<- error,"Rubbish not found"
<- (gdb)
```



## 19.2 CLI に対する GDB/MI の互換性

GDB の既存の CLI インターフェイスを熟知しているユーザを支援するために、GDB/MI は既存の CLI コマンドも受け付けます。構文によって指定されているとおり、そのようなコマンドは GDB/MI インターフェイスに対して直接入力することが可能であり、それに対して GDB が応答を返します。

このメカニズムは、GDB/MI クライアントの開発者を支援するために提供されているもので、CLI に対する信頼性のあるインターフェイスとして提供されているわけではありません。コマンドは、GDB/MI が動作していることを前提とする環境において解釈されるため、実際の出力は、GDB/MI の出力と CLI の出力とが混在したものになってしまうでしょう。このような混在出力をサポートしているわけではありません。

## 19.3 GDB/MI 出力レコード

### 19.3.1 GDB/MI 結果レコード

いくつかの帯域外通知<sup>2</sup>に加えて、GDB/MI コマンドに対する応答には、以下のような結果の表示が 1 つ含まれます。

`"^done" [ ",", " results ]`

同期オペレーションが成功したことを示します。*results*が戻り値です。

`"^running"`

非同期オペレーションの開始が成功したことを示します。ターゲットは実行中です。

`"^error" ", " c-string`

オペレーションが失敗したことを示します。*c-string*には、対応するエラー・メッセージが含まれます。

### 19.3.2 GDB/MI ストリーム・レコード

GDB は、内部的にいくつかの出力ストリームを保持しています。コンソール用、ターゲット用、ログ用のものです。個々のストリームに向けた出力は、ストリーム・レコードを使って GDB/MI インターフェイスを経由します。

個々のストリーム・レコードは、そのストリームを一意に識別する接頭文字で始まります（セクション 19.1.2 [GDB/MI 出力構文], ページ 180 参照）。この接頭文字に加えて、個々のストリーム・レコードは *string-output* を含みます。これは、（暗黙の改行を含む）加工されていないテキストか、もしくは、引用符で囲まれた C 文字列（こちらは暗黙の改行を含みません）のいずれかです。

`"~" string-output`

コンソール出力ストリームには、CLI のコンソール・ウィンドウに表示されるべきテキストが入ります。そこには、CLI コマンドへのテキストによる応答が入ります。

`"@" string-output`

ターゲット出力ストリームには、実行中のターゲットからのテキストによる出力が入ります。

<sup>2</sup> 訳注：セクション 19.3.3 [GDB/MI 帯域外レコード], ページ 184 参照

"&" *string-output*

ログ・ストリームには、GDB 内部から出力されるデバッグ用メッセージが入ります。

### 19.3.3 GDB/MI 帯域外レコード

帯域外レコードは、追加的に発生した変化を GDB/MI クライアントに通知するために使われます。これらの変化は、GDB/MI の結果（例えばブレイクポイントの変更）、または、ターゲットの活動結果（例えばターゲットの停止）のいずれかです。

以下に、可能な帯域外レコードの予備的な一覧を示します。

"\*" "stop"

## 19.4 GDB/MI コマンド記述フォーマット

残りのセクションでは、コマンド・ブロックについて記述します。個々のコマンド・ブロックは、この章と類似の方法でレイアウトされています。

例の中に示されている改行は読みやすくするためのものであるという点に注意してください。実際の出力では改行されません。また、( N.A. と記されている ) 例のないコマンドはまだ実装されていませんので、その点にも注意してください。

### 動機

このコマンド集合を提供する動機

### 紹介

このコマンド集合全体の簡単な紹介

### コマンド

ブロック内の個々のコマンドについて、以下のことが説明されます。

### 概要

`-command args...`

## GDB コマンド

対応する GDB CLI コマンド

### 結果

### 帯域外

### 注

### 例

## 19.5 GDB/MI ブレイクポイント・テーブル・コマンド

このセクションでは、ブレイクポイントを操作するための GDB/MI コマンドについて説明します。

### -break-after コマンド

#### 概要

`-break-after number count`

ブレイクポイント番号が *number* のブレイクポイントは、そこに *count* 回到達した後にしか有効になりません。このことが ‘-break-list’ コマンドの出力にどのように反映されるかという点については、下記の ‘-break-list’ コマンドの説明を参照してください。

#### GDB コマンド

対応する GDB コマンドは ‘ignore’ です。

#### 例

```
(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)
```

### -break-condition

#### 概要

`-break-condition number expr`

ブレイクポイント番号が *number* のブレイクポイントは、*expr* によって指定される条件が真である場合にのみ停止します。この条件は、‘-break-list’ の出力の一部となります (下記の ‘-break-list’ コマンドの説明を参照してください)。

#### GDB コマンド

対応する GDB コマンドは ‘condition’ です。

例

```
(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)
```

### `-break-delete` コマンド

概要

```
-break-delete ( breakpoint )+
```

引数リストにおいて指定されたブレイクポイント番号 ( 複数可 ) を持つブレイクポイントを削除します。これは当然、ブレイクポイント一覧に反映されます。

### GDB コマンド

対応する GDB コマンドは `'delete'` です。

例

```
(gdb)
-break-delete 1
^done
(gdb)
-break-list
^done,BreakpointTable={}
(gdb)
```

### `-break-disable` コマンド

概要

```
-break-disable ( breakpoint )+
```

*breakpoint* により指定されたブレイクポイント ( 複数可 ) を無効にします。ここで指定されたブレイクポイントについては、ブレイクポイント一覧の `'enabled'` フィールドに `'n'` がセットされます。

### GDB コマンド

対応する GDB コマンドは `'disable'` です。

例

```
(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)
```

**-break-enable** コマンド

概要

```
-break-enable ( breakpoint )+
```

*breakpoint* により指定される ( 以前に無効にされた ) ブレイクポイント ( 複数可 ) を有効にします。

## GDB コマンド

対応する GDB コマンドは ‘enable’ です。

例

```
(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)
```

**-break-info** コマンド

概要

```
-break-info breakpoint
```

1 つのブレイクポイントに関する情報を取得します。

## GDB コマンド

対応する GDB コマンドは ‘info break *breakpoint*’ です。

例

N.A.

`-break-insert` コマンド

概要

```
-break-insert [ -t ] [ -h ] [ -r ]
               [ -c condition ] [ -i ignore-count ]
               [ -p thread ] [ line | addr ]
```

*line* が指定される場合、それは以下のいずれか 1 つです。

- 関数名
- ファイル名:行番号
- ファイル名:関数名
- \*アドレス

このコマンドにおいて使うことのできる任意選択のパラメータには以下があります。

‘-t’            一時的なブレイクポイントを挿入します。

‘-h’            ハードウェア・ブレイクポイントを挿入します。

‘-c *condition*’  
                ブレイクポイントに条件 *condition* を付与します。

‘-i *ignore-count*’  
                *ignore-count* を初期化します。

‘-r’            与えられた正規表現にマッチする名前を持つ関数すべてに、通常のブレイクポイントを挿入します。これ以外のフラグは、正規表現には適用できません。

結果

結果は以下のような形式になります。

```
^done,bkptno="number",func="funcname",
file="filename",line="lineno"
```

*number* は、このブレイクポイントに対して GDB が付与した番号です。*funcname* は、ブレイクポイントが挿入された関数の名前です。*filename* は、この関数を含むソース・ファイルの名前です。*lineno* は、そのファイル内のソースの行番号です。

注: このフォーマットは変更される可能性があります。

GDB コマンド

対応する GDB コマンドは、‘break’、‘tbreak’、‘hbreak’、‘thbreak’、‘rbreak’です。

例

```
(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)
```

## -break-list コマンド

概要

```
-break-list
```

挿入されたブレイクポイントの一覧を表示します。表示される一覧には以下のフィールドが含まれます。

‘Number’     ブレイクポイント番号

‘Type’       ブレイクポイント種別: ‘breakpoint’または‘watchpoint’

‘Disposition’  
ブレイクポイントに到達したときに、それを削除するべきか、あるいは、無効にするべきか: ‘keep’または‘nokeep’

‘Enabled’    ブレイクポイントは有効か: ‘y’または‘n’

‘Address’    ブレイクポイントがセットされたメモリ位置

‘What’       関数名、ファイル名、行番号によって表現される、ブレイクポイントの論理的な位置

‘times’      ブレイクポイントの到達回数

ブレイクポイントやウォッチポイントが存在しない場合、BreakpointTable フィールドの一覧は空になります。

## GDB コマンド

対応する GDB コマンドは ‘info break’ です。

例

```
(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)
```

以下に、ブレイクポイントが存在しない場合の結果の例を示します。

```
(gdb)
-break-list
^done,BreakpointTable={}
(gdb)
```

## -break-watch コマンド

概要

```
-break-watch [ -a | -r ]
```

ウォッチポイントを生成します。‘-a’オプションとともに指定すると、*access* ウォッチポイントが生成されます。これは、あるメモリ位置に対する読み取りや書き込みが行われたことを契機とするウォッチポイントです。‘-r’オプションとともに指定すると、*read* ウォッチポイントが生成されます。これは、あるメモリ位置が読み取りのためにアクセスされたことのみを契機とするウォッチポイントです。どちらのオプションも指定しないと、生成されるのは通常のウォッチポイントです。これは、あるメモリ位置が書き込みのためにアクセスされたことを契機とするウォッチポイントです。セクション 5.1.2 [ウォッチポイントの設定], ページ 36 を参照してください。

‘-break-list’は、挿入されたウォッチポイントとブレイクポイントを1つの一覧にして報告しますので、注意してください。

## GDB コマンド

対応する GDB コマンドは、‘watch’、‘awatch’、‘rwatch’です。

例

main関数の中の変数に対してウォッチポイントをセットした例を示します。

```
(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)

ある関数の中の局所的な変数に対してウォッチポイントをセットした例を示します。GDB はプログラムの実行を 2 回停止します。1 回目はその変数の値の変更によるもの、2 回目はウォッチポイントがスコープの範囲外になることによるものです。

```
(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)
```

ブレイクポイントとウォッチポイントの一覧を、プログラム実行のさまざまな時点において表示させた場合の例を示します。ウォッチポイントがスコープの範囲外になると一覧から削除されることに注意してください。

```
(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 GDB/MI データ操作

このセクションでは、データを操作する GDB/MI コマンドについて説明します。データの操作とは、メモリやレジスタの値を調べたり、式を評価することなどを指します。

### -data-disassemble コマンド

#### 概要

```

-data-disassemble
  [ -s start-addr -e end-addr ]
  | [ -f filename -l linenum [ -n lines ] ]
  -- mode

```

各項目について以下に説明します。

- ‘start-addr’ 先頭アドレス ( \$pc の値 )
- ‘end-addr’ 終端アドレス
- ‘filename’ 逆アセンブルするファイルの名前
- ‘linenum’ その前後を逆アセンブルする行の番号
- ‘lines’ 生成される逆アセンブル行の行数。この値が -1 で、end-addr が指定されていない場合、関数全体が逆アセンブルされます。end-addr がゼロ以外の値として指定されていて、start-addr と end-addr の間の逆アセンブル行数よりも lines の値が小さい場合、lines により指定される行数しか表示されません。lines の値が、start-addr と end-addr の間の行数よりも大きい場合は、end-addr までの行しか表示されません。
- ‘mode’ 0 ( 逆アセンブル結果のみの表示を指定 ) または 1 ( ソースと逆アセンブル結果を混在させた表示を指定 ) のいずれかです。

## 結果

個々の命令に対応する出力は 2 つのフィールドから構成されます。

- Address ( アドレス )
- Func-name ( 関数名 )
- Offset ( オフセット )
- Instruction ( 命令 )

命令フィールドに含まれる情報は、flathead によって直接操作されることはないという点に注意してください。すなわち、そのフォーマットを調整することは不可能であるということです。

## GDB コマンド

このコマンドからの CLI に対する直接のマッピングは存在しません。

## 例

\$pc から \$pc + 20 までの範囲の現在の値を逆アセンブルする例を示します。

```
(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)
```

main 関数全体を逆アセンブルする例を示します。行番号 32 は main の中にあります。

```
-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)
```

main の先頭から 3 つの命令を逆アセンブルする例を示します。

```
(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)
```

mainの先頭から 3 つの命令を逆アセンブルして、混在モードで表示する例を示します。

```
(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)
```

## -data-evaluate-expressionコマンド

### 概要

-data-evaluate-expression *expr*

*expr* を式として評価します。その式の中には下位の関数呼び出しを含めることもできます。関数呼び出しは同期的に実行されます。式に空白が含まれる場合は、式を二重引用符で囲む必要があります。

## GDB コマンド

対応する GDB コマンドは、‘print’、‘output’、‘call’です。gdbtkにのみ、対応するコマンドとして‘gdb\_eval’があります。

### 例

以下の例においてコマンドに先行する番号は、セクション 19.1 [GDB/MI コマンド構文], ページ 179 に説明されているトークンです。GDB/MI が、その出力の中で同一のトークンを返していることに注目してください。

```

211-data-evaluate-expression A
211^done,value="1"
(gdb)
311-data-evaluate-expression &A
311^done,value="0xefffef7c"
(gdb)
411-data-evaluate-expression A+3
411^done,value="4"
(gdb)
511-data-evaluate-expression "A + 3"
511^done,value="4"
(gdb)

```

## -data-list-changed-registers コマンド

### 概要

```
-data-list-changed-registers
```

値の変化したレジスタの一覧を表示します。

### GDB コマンド

このコマンドに直接類似するものを GDB は提供していません。gdbtk には、対応するコマンドとして 'gdb\_changed\_register\_list' があります。

### 例

PPC MBX ボード 上における例を示します。

```

(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)

```

## -data-list-register-names コマンド

### 概要

```
-data-list-register-names [ ( regno )+ ]
```

カレント・ターゲットのレジスタ名の一覧を表示します。引数が指定されない場合、すべてのレジスタの名前を表示します。正数値が引数として指定された場合、その引数に対応するレジスタの名前の一覧を表示します。

## GDB コマンド

‘-data-list-register-names’に対応するコマンドを GDB は提供していません。gdbtkには、対応するコマンドとして‘gdb\_regnames’があります。

例

PPC MBX ボード上における例を示します。

```
(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)
```

## -data-list-register-values コマンド

概要

```
-data-list-register-values fmt [ ( regno )*]
```

レジスタの内容を表示します。*fmt* によって指定されるフォーマットにしたがって、レジスタの内容が表示されます。必須ではありませんが、*fmt* の後ろに、表示すべきレジスタを指定する番号の一覧を続けることもできます。番号一覧を指定しないと、すべてのレジスタの内容を表示しなければならないことを指定したことになります。

*fmt* に指定可能なフォーマットには以下のものがあります。

|   |                              |
|---|------------------------------|
| x | 16 進数                        |
| o | 8 進数                         |
| t | 2 進数                         |
| d | 10 進数                        |
| r | バイナリ・データ (Raw)               |
| N | ナチュラル (natural) <sup>3</sup> |

<sup>3</sup> 訳注：変数の型に応じて自動的に選択されるデフォルトのフォーマット

## GDB コマンド

対応する GDB コマンドは、‘info reg’、‘info all-reg’です。また、gdbtkにおいては‘gdb\_fetch\_registers’です。

### 例

PPC MBX ボード上における例を示します。(注: 改行は読みやすくするためのものであるという点に注意してください。実際の出力では改行されません)

```
(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="0xffffffffe"},
{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="0xffdffff"},
{number="15",value="0xffffffff"},{number="16",value="0xffffffe"},
{number="17",value="0xefffffff"},{number="18",value="0xffffffe"},
{number="19",value="0xffffffff"},{number="20",value="0xffffffff"},
{number="21",value="0xffffffff"},{number="22",value="0xffffffff7"},
{number="23",value="0xffffffff"},{number="24",value="0xffffffff"},
{number="25",value="0xffffffff"},{number="26",value="0xfffffff"},
{number="27",value="0xffffffff"},{number="28",value="0xf7bffff"},
{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)
```

## -data-read-memory コマンド

### 概要

```
-data-read-memory [ -o byte-offset ]
    address word-format word-size
    nr-rows nr-cols [ aschar ]
```

各項目について以下に説明します。

‘*address*’    読み取るべき最初のメモリ・ワードのアドレスを指定する式です。空白を含む複雑な式は、C の慣例にしたがって引用符で囲むべきです。

‘*word-format*’    メモリ・ワードを表示する際に使用するフォーマットです。その表記法は、GDB の `print` コマンドと同一です ( セクション 8.4 [出力フォーマット], ページ 66 参照 )。

‘*word-size*’    個々のメモリ・ワードのバイト単位のサイズです。

‘*nr-rows*’    出力される表の行数です。

‘*nr-cols*’    出力される表の列数です。

‘*aschar*’    これが指定されると、各行には ASCII ダンプが含まれるべきことを示します。*aschar* の値は、あるバイトが表示可能な ASCII 文字セットのメンバではない場合に、パディング用の文字として使われます ( 表示可能な ASCII 文字とは、そのコードが 32 以上 126 以下の文字を指します )。

‘*byte-offset*’    メモリの内容を獲得する前に *address* に対して加算すべきオフセットです。

このコマンドは、*nr-rows* 行 *nr-cols* 列のワードから構成されるテーブルとしてメモリ内容を表示します。個々のワードのサイズは *word-size* バイトです。全体では *nr-rows* \* *nr-cols* \* *word-size* により計算されるバイト ( これが ‘*total-bytes*’ として表示されます ) が読み取られます。指定されたバイト数よりも少ない情報がターゲットによって返された場合は、存在しないワードは ‘N/A’ によって示されます。ターゲットによって読み取られたバイト数は ‘*nr-bytes*’ によって、また、メモリの読み取りの際の先頭アドレスは ‘*addr*’ によって、それぞれ示されます。

次の行のアドレスは ‘*next-row*’、前の行のアドレスは ‘*prev-row*’、次のページのアドレスは ‘*next-page*’ 前のページのアドレスは ‘*prev-page*’ において、それぞれ示されます。

## GDB コマンド

対応する GDB コマンドは ‘*x*’ です。また `gdbtk` では、‘`gdb_get_mem`’ によるメモリの読み取りが提供されています。



例

bytes+6を先頭とするメモリから 6 バイトを読み取り、その後-6バイトだけオフセットする例を以下に示します。3 行 2 列、1 ワードあたり 1 バイト、個々のワードを 16 進としてフォーマットします。

```
(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)
```

アドレス shorts + 64を先頭とするメモリから 2 バイトを読み取り、10 進数としてフォーマットされた単一のワードとして表示する例を以下に示します。

```
(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)
```

bytes+16を先頭とするメモリから 32 バイトを読み取り、8 行 4 列にフォーマットする例を以下に示します。表示不可の文字は x で置き換えてエンコードした文字列を含みます。

```
(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)
```

## -display-delete コマンド

概要

```
-display-delete number
```

*number* によって指定される表示を削除します。

## GDB コマンド

対応する GDB コマンドは ‘delete display’ です。

例

N.A.

-display-disable コマンド

概要

`-display-disable number`

*number* によって指定される表示を無効にします。

## GDB コマンド

対応する GDB コマンドは ‘disable display’ です。

例

N.A.

-display-enable コマンド

概要

`-display-enable number`

*number* によって指定される表示を有効にします。

## GDB コマンド

対応する GDB コマンドは ‘enable display’ です。

例

N.A.

-display-insert コマンド

概要

`-display-insert expression`

プログラムが停止するたびに *expression* を表示します。

## GDB コマンド

対応する GDB コマンドは ‘display’ です。

例

N.A.

`-display-list` コマンド

概要

`-display-list`

表示対象の一覧を表示します。カレントな値を表示することはありません。

## GDB コマンド

対応する GDB コマンドは `'info display'` です。

例

N.A.

`-environment-cd` コマンド

概要

`-environment-cd pathdir`

GDB の作業ディレクトリをセットします。

## GDB コマンド

対応する GDB コマンドは `'cd'` です。

例

```
(gdb)
-environment-cd /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done
(gdb)
```

`-environment-directory` コマンド

概要

`-environment-directory pathdir`

ソース・ファイルの探索パスの先頭に、ディレクトリ *pathdir* を追加します。

## GDB コマンド

対応する GDB コマンドは `'dir'` です。

例

```
(gdb)
-environment-directory /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done
(gdb)
```

`-environment-path` コマンド

概要

```
-environment-path ( pathdir )+
```

オブジェクト・ファイルの探索パスの先頭に、ディレクトリ (複数可) を追加します。

## GDB コマンド

対応する GDB コマンドは ‘path’ です。

例

```
(gdb)
-environment-path /kwikemart/marge/ezannoni/flathead-dev/ppc-eabi/gdb
^done
(gdb)
```

`-environment-pwd` コマンド

概要

```
-environment-pwd
```

カレントな作業ディレクトリを表示します。

## GDB コマンド

対応する GDB コマンドは ‘pwd’ です。

例

```
(gdb)
-environment-pwd
~Working directory /kwikemart/marge/ezannoni/flathead-dev/devo/gdb.
^done
(gdb)
```

## 19.7 GDB/MI プログラム制御

## プログラム終了

下位のプログラムがブレイクポイントに一切到達しなかった場合、実行の帰結として、完全に終了してしまうことがあります。このような場合に、プログラムが例外的な終了を行ったのであれば、出力には終了コードが含まれます。

例:

プログラムが正常に終了した場合:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited-normally"
(gdb)
```

プログラムが例外的に終了した場合:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited",exit-code="01"
(gdb)
```

これ以外にも、SIGINTのようなシグナルを受信してプログラムが終了することがあります。この場合には、GDB/MI は以下のように出力します。

```
(gdb)
*stopped,reason="exited-signalled",signal-name="SIGINT",
signal-meaning="Interrupt"
```

## -exec-abort コマンド

### 概要

```
-exec-abort
```

実行中の下位プログラムを強制終了 ( kill ) します。

## GDB コマンド

対応する GDB コマンドは 'kill' です。

例

N.A.

## -exec-arguments コマンド

### 概要

```
-exec-arguments args
```

今回の ‘-exec-run’ において下位プログラムによって使われる引数をセットします。

### GDB コマンド

対応する GDB コマンドは ‘set args’ です。

### 例

なし。

### -exec-continue コマンド

### 概要

```
-exec-continue
```

非同期コマンド。下位プログラムの実行を再開し、ブレイクポイントに到達するか終了するまで実行を継続します。

### GDB コマンド

対応する GDB コマンドは ‘continue’ です。

### 例

```
-exec-continue
^running
(gdb)
@Hello world
*stopped,reason="breakpoint-hit",bkptno="2",frame={func="foo",args={}},
file="hello.c",line="13"}
(gdb)
```

### -exec-finish コマンド

### 概要

```
-exec-finish
```

非同期コマンド。下位プログラムの実行を再開し、カレントな関数が終了するまで実行を継続します。関数の戻り値を表示します。

### GDB コマンド

対応する GDB コマンドは ‘finish’ です。

## 例

戻り値が `void` の関数の例を以下に示します。

```
-exec-finish
^running
(gdb)
@hello from foo
*stopped,reason="function-finished",frame={func="main",args={},
file="hello.c",line="7"}
(gdb)
```

戻り値が `void` 以外の関数の例を以下に示します。戻り値を保持している GDB の内部変数の名前が、その値とともに表示されます。

```
-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)
```

## `-exec-interrupt` コマンド

## 概要

```
-exec-interrupt
```

非同期コマンド。ターゲットのバックグラウンドにおける実行に割り込みます。停止メッセージに関連付けられているトークン<sup>4</sup>が、割り込まれた実行コマンドのトークンである点に注目してください。割り込み自体のトークンは、`^done` という出力においてのみ現われます。ユーザが、実行中ではないプログラムに割り込もうとすると、エラー・メッセージが表示されることになります。

## GDB コマンド

対応する GDB コマンドは `interrupt` です。

## 例

```
(gdb)
111-exec-continue
111^running

(gdb)
222-exec-interrupt
222^done
(gdb)
```

---

<sup>4</sup> 訳注：下記の例における 111 や 222

```
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)
```

## `-exec-next` コマンド

### 概要

```
-exec-next
```

非同期コマンド。下位プログラムの実行を再開し、次のソース行の先頭に到達したところで停止します。

### GDB コマンド

対応する GDB コマンドは `'next'` です。

### 例

```
-exec-next
^running
(gdb)
*stopped,reason="end-stepping-range",line="8",file="hello.c"
(gdb)
```

## `-exec-next-instruction` コマンド

### 概要

```
-exec-next-instruction
```

非同期コマンド。マシン命令を 1 つ実行します。その命令が関数呼び出しである場合は、関数が復帰するまで実行を継続します。プログラムがソース行の途中に対応する命令で停止した場合には、そのアドレスも表示されます。

### GDB コマンド

対応する GDB コマンドは `'nexti'` です。

### 例

```
(gdb)
-exec-next-instruction
^running
```



```
(gdb)
*stopped,reason="end-stepping-range",
addr="0x000100d4",line="5",file="hello.c"
(gdb)
```

## `-exec-return` コマンド

### 概要

`-exec-return`

カレントな関数を即時に復帰させます。下位プログラムを実行することはありません。新しくカレントとなったフレームを表示します。

### GDB コマンド

対応する GDB コマンドは `'return'` です。

### 例

```
(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)
```

## `-exec-run` コマンド

### 概要

`-exec-run`

非同期コマンド。下位プログラムの先頭から実行を開始します。下位プログラムは、ブレイクポイントに到達するかプログラムが終了するまで実行されます。

## GDB コマンド

対応する GDB コマンドは ‘run’ です。

例

```
(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)
```

## -exec-show-arguments コマンド

概要

```
-exec-show-arguments
```

プログラムの引数を表示します。

## GDB コマンド

対応する GDB コマンドは ‘show args’ です。

例

N.A.

## -exec-step コマンド

概要

```
-exec-step
```

非同期コマンド。下位プログラムの実行を再開し、次のソース行が関数呼び出しでなければ、次のソース行の先頭で停止します。次のソース行が関数呼び出しの場合は、呼び出される関数の最初の命令のところで停止します。

## GDB コマンド

対応する GDB コマンドは ‘step’ です。

## 例

関数の内部に入り込むステップ実行の例を以下に示します。

```
-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)
```

通常のステップ実行の例を以下に示します。

```
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",line="14",file="recursive2.c"
(gdb)
```

### -exec-step-instruction コマンド

## 概要

```
-exec-step-instruction
```

非同期コマンド。下位プログラムの実行を再開して、マシン命令を 1 つ実行します。GDB がプログラムを停止する際の出力は、停止したところがソース行の途中でどうかに依存して変わります。ソース行の途中の場合は、プログラムが停止したところのアドレスも表示されます。

## GDB コマンド

対応する GDB コマンドは 'stepi' です。

## 例

```
(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)
```

## `-exec-until` コマンド

### 概要

`-exec-until [ location ]`

非同期コマンド。引数で指定された *location* まで下位プログラムを実行します。引数が指定されない場合、下位プログラムは、カレント行よりも後ろの行に到達するまで実行されます。この場合、`reason` 項目の値は “location-reached” になります。

## GDB コマンド

対応する GDB コマンドは ‘until’ です。

### 例

```
(gdb)
-exec-until recursive2.c:6
^running
(gdb)
x = 55
*stopped,reason="location-reached",frame={func="main",args={}},
file="recursive2.c",line="6"}
(gdb)
```

## `-file-exec-and-symbols` コマンド

### 概要

`-file-exec-and-symbols file`

デバッグ対象となる実行ファイルを指定します。このファイルからシンボル・テーブルも読み込まれます。ファイルが指定されないと、このコマンドは、実行ファイル情報とシンボル情報をクリアします。このコマンドが引数を指定せずに使われたときにブレイクポイントがセットされていると、GDB はエラー・メッセージを出力します。これ以外の場合には、完了通知だけが出力され、それ以外には何も出力されません。

## GDB コマンド

対応する GDB コマンドは ‘file’ です。

### 例

```
(gdb)
-file-exec-and-symbols /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

## `-file-exec-file` コマンド

### 概要

`-file-exec-file file`

デバッグ対象となる実行ファイルを指定します。‘`-file-exec-and-symbols`’とは異なり、このファイルからシンボル・テーブルの読み込みは行われません。引数を指定せずに使われると、GDB は実行ファイルに関する情報をクリアします。完了通知以外には何も出力されません。

### GDB コマンド

対応する GDB コマンドは ‘`exec-file`’ です。

### 例

```
(gdb)
-file-exec-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

`-file-list-exec-sections` コマンド

### 概要

`-file-list-exec-sections`

カレントな実行ファイルのセクションの一覧を表示します。

### GDB コマンド

GDB の ‘`info file`’ コマンドなどが、このコマンドと同一の情報を表示します。gdbtk には、対応するコマンドとして ‘`gdb_load_info`’ があります。

### 例

N.A.

`-file-list-exec-source-files` コマンド

### 概要

`-file-list-exec-source-files`

カレントな実行ファイルのソース・ファイルの一覧を表示します。

### GDB コマンド

これに直接対応する GDB コマンドは存在しません。gdbtk は、類似のコマンドとして ‘`gdb_listfiles`’ を提供しています。

### 例

N.A.

## `-file-list-shared-libraries` コマンド

### 概要

`-file-list-shared-libraries`

プログラムが使用する共用ライブラリの一覧を表示します。

### GDB コマンド

対応する GDB コマンドは `'info shared'` です。

### 例

N.A.

## `-file-list-symbol-files` コマンド

### 概要

`-file-list-symbol-files`

シンボル・ファイルの一覧を表示します。

### GDB コマンド

対応する GDB コマンドは `'info file'` (の一部の機能) です。

### 例

N.A.

## `-file-symbol-file` コマンド

### 概要

`-file-symbol-file file`

指定された *file* 引数 (の指すファイル) からシンボル・テーブルを読み込みます。引数を指定せずに使われると、GDB のシンボル・テーブル情報をクリアします。完了通知以外には何も出力されません。

### GDB コマンド

対応する GDB コマンドは `'symbol-file'` です。

### 例

```
(gdb)
-file-symbol-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

## 19.8 GDB/MI におけるその他の GDB コマンド

### -gdb-exit コマンド

#### 概要

```
-gdb-exit
```

直ちに GDB を終了させます。

#### GDB コマンド

‘quit’にほぼ対応します。

#### 例

```
(gdb)
-gdb-exit
```

### -gdb-set コマンド

#### 概要

```
-gdb-set
```

GDB の内部変数に値をセットします。

#### GDB コマンド

対応する GDB コマンドは ‘set’です。

#### 例

```
(gdb)
-gdb-set $foo=3
^done
(gdb)
```

### -gdb-show コマンド

#### 概要

```
-gdb-show
```

GDB 変数のカレントな値を表示します。

#### GDB コマンド

対応する GDB コマンドは ‘show’です。

例

```
(gdb)
-gdb-show annotate
^done,value="0"
(gdb)
```

`-gdb-version` コマンド

概要

```
-gdb-version
```

GDB のバージョン情報を表示します。ほとんどの場合、テストにおいて使われます。

## GDB コマンド

同等の GDB コマンドは存在しません。GDB はデフォルトでは、対話セッションを開始したときにこの情報を表示します。

例

```
(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 GDB/MI におけるスタック操作コマンド

`-stack-info-frame` コマンド

概要

```
-stack-info-frame
```

カレントなフレームに関する情報を取得します。

## GDB コマンド

対応する GDB コマンドは `'info frame'` または (引数を指定しない) `'frame'` です



例

N.A.

`-stack-info-depth` コマンド

概要

```
-stack-info-depth [ max-depth ]
```

スタックの深さを返します。整数引数 *max-depth* が指定されると、*max-depth* 個を超えるフレームを数えることをしません。

## GDB コマンド

対応する GDB コマンドは存在しません。

例

フレーム・レベル 0 からフレーム・レベル 11 までを持つスタックに対して実行した例を以下に示します。

```
(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)
```

`-stack-list-arguments` コマンド

概要

```
-stack-list-arguments show-values
[ low-frame high-frame ]
```

(フレーム・レベルが) *low-frame* 以上 *high-frame* 以下のフレームの引数リストを表示します。*low-frame* と *high-frame* が指定されないと、すべての呼び出しスタックについて引数の一覧を表示します。

*show-values* 引数の値は 0 または 1 のいずれかの値を取らなければなりません。0 の場合は、引数の名前の一覧だけが表示されることを意味し、1 の場合は、引数の名前と値の両方が表示されることを意味します。

## GDB コマンド

同等のコマンドは GDB には存在しません。gdbtk には 'gdb\_get\_args' というコマンドがあり、その機能は '-stack-list-arguments' の機能と一部重複しています。

例

```
(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 \"A string argument.\""}},
  frame={level="2", args={
  {name="intarg", value="2"},
  {name="strarg", value="0x11940 \"A string argument.\""}},
  frame={level="3", args={
  {name="intarg", value="2"},
  {name="strarg", value="0x11940 \"A string argument.\""},
  {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 \"A string argument.\""}}}}
(gdb)
```

## -stack-list-frames コマンド

### 概要

```
-stack-list-frames [ low-frame high-frame ]
```

その時点においてスタック上に存在するフレームの一覧を表示します。個々のフレームについて、以下の情報が表示されます。

‘level’ フレーム番号。最下位のフレーム、すなわち、最下位の関数が 0 になります。

‘addr’ そのフレームの \$pc<sup>5</sup> の値。

‘func’ 関数の名前。

‘file’ 関数が存在するソース・ファイルの名前。

‘line’ \$pc に対応する行番号。

引数を指定せずに実行されると、このコマンドは全スタックのバックトレースを表示します。整数引数が 2 つ指定されると、レベルがその 2 つの数の間 ( それらの数を含む ) にあるフレームを表示します。2 つの引数の値が等しい場合、対応するレベルのフレームだけが表示されます。

## GDB コマンド

対応する GDB コマンドは ‘backtrace’ と ‘where’ です。

### 例

スタック全体のバックトレースの例を以下に示します。

```
(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"},
```

---

<sup>5</sup> 訳注：プログラム・カウンタ

```

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)

```

2つのフレームレベルの間にあるフレームを表示する例を以下に示します。

```

(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)

```

単一のフレームを表示する例を以下に示します。

```

(gdb)
-stack-list-frames 3 3
^done,stack=
{frame={level="3 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"}}
(gdb)

```

## `-stack-list-locals` コマンド

### 概要

```
-stack-list-locals print-values
```

カレントなフレームにおける局所変数の名前を表示します。引数に 0 が指定されると変数の名前だけが表示されます。引数に 1 が指定されるとその値も表示されます。

## GDB コマンド

GDB においては `'info locals'`、gdbtk においては `'gdb_get_locals'` が、それぞれ対応するコマンドです。

### 例

```
(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"}}
```

#### `-stack-select-frame` コマンド

##### 概要

```
-stack-select-frame framenum
```

カレントなフレームを替えます。スタック上に存在する、*framenum* で指定される番号を持つ別フレームを選択します。

##### GDB コマンド

対応する GDB コマンドは、`'frame'`、`'up'`、`'down'`、`'select-frame'`、`'up-silent'`、`'down-silent'` です。

##### 例

```
(gdb)
-stack-select-frame 2
^done
(gdb)
```

## 19.10 GDB/MI のシンボル・クエリー・コマンド

#### `-symbol-info-address` コマンド

##### 概要

```
-symbol-info-address symbol
```

*symbol* により指定されるシンボルがどこに格納されているかを示します。

##### GDB コマンド

対応する GDB コマンドは `'info address'` です。

##### 例

N.A.

#### `-symbol-info-file` コマンド

### 概要

`-symbol-info-file`

シンボルに対応するファイルを示します。

### GDB コマンド

対応する GDB コマンドは存在しません。gdbtkにおいては `'gdb_find_file'` が対応するコマンドです。

### 例

N.A.

`-symbol-info-function` コマンド

### 概要

`-symbol-info-function`

シンボルが存在する関数を示します。

### GDB コマンド

gdbtkにおける `'gdb_get_function'` が対応するコマンドです。

### 例

N.A.

`-symbol-info-line` コマンド

### 概要

`-symbol-info-line`

ソース行に対応するコードのコア・アドレスを示します。

### GDB コマンド

対応する GDB コマンドは `'info line'` です。gdbtkでは、`'gdb_get_line'`、`'gdb_get_file'` です。

### 例

N.A.

`-symbol-info-symbol` コマンド

### 概要

```
-symbol-info-symbol addr
```

*addr* により指定される位置にあるシンボルを示します。

### GDB コマンド

対応する GDB コマンドは 'info symbol' です。

### 例

N.A.

### -symbol-list-functions コマンド

### 概要

```
-symbol-list-functions
```

実行ファイルの中の関数の一覧を表示します。

### GDB コマンド

GDB においては 'info functions'、gdbtk においては 'gdb\_listfunc' と 'gdb\_search' が、それぞれ対応するコマンドです。

### 例

N.A.

### -symbol-list-types コマンド

### 概要

```
-symbol-list-types
```

すべての型の名前の一覧を表示します。

### GDB コマンド

GDB においては 'info types'、gdbtk においては 'gdb\_search' が、それぞれ対応するコマンドです。

### 例

N.A.

### -symbol-list-variables コマンド

### 概要

`-symbol-list-variables`

すべての広域変数と静的変数の名前の一覧を表示します。

### GDB コマンド

GDB においては `'info variables'`、gdbtk においては `'gdb_search'` が、それぞれ対応するコマンドです。

### 例

N.A.

### `-symbol-locate` コマンド

### 概要

`-symbol-locate`

### GDB コマンド

gdbtk における `'gdb_loc'` が対応するコマンドです。

### 例

N.A.

### `-symbol-type` コマンド

### 概要

`-symbol-type variable`

*variable* により指定される変数の型を示します。

### GDB コマンド

対応する GDB コマンドは `'ptype'` です。gdbtk においては、`'gdb_obj_variable'` が対応するコマンドです。

### 例

N.A.



## 19.11 GDB/MI のターゲット操作コマンド

### -target-attach コマンド

#### 概要

`-target-attach pid | file`

GDB の外部に存在する、*pid* により指定されるプロセス ID を持つプロセス、または、*file* により指定されるファイルにアタッチします。

#### GDB コマンド

対応する GDB コマンドは 'attach' です。

#### 例

N.A.

### -target-compare-sections コマンド

#### 概要

`-target-compare-sections [ section ]`

ターゲットの *section* により指定されるセクションのデータと実行ファイルを比較します。引数を指定しないと、すべてのセクションが比較されます。

#### GDB コマンド

GDB における同等のコマンドは 'compare-sections' です。

#### 例

N.A.

### -target-detach コマンド

#### 概要

`-target-detach`

リモート・ターゲットへの接続を切断します。出力はありません。

#### GDB コマンド

対応する GDB コマンドは 'detach' です。

例

```
(gdb)
-target-detach
^done
(gdb)
```

-target-download コマンド

概要

-target-download

実行ファイルをリモート・ターゲットにロードします。0.5 秒ごとに更新メッセージを表示します。このメッセージには、以下のフィールドが含まれます。

‘section’ セクションの名前。

‘section-sent’  
そのセクションの送信済みのサイズ。

‘section-size’  
セクションのサイズ。

‘total-sent’  
( カレントなセクションと以前のセクションを合わせた ) 送信済みの総サイズ。

‘total-size’  
ダウンロードすべきすべての実行ファイルのサイズ。

個々のメッセージはステータス・レコードとして送信されます ( セクション 19.1.2 [GDB/MI 出力構文], ページ 180 参照 )。

さらに、個々のセクションの名前とサイズを、それらがダウンロードされる際に表示します。このメッセージには、以下のフィールドが含まれます。

‘section’ セクションの名前。

‘section-size’  
セクションのサイズ。

‘total-size’  
ダウンロードすべきすべての実行ファイルのサイズ。

最後に要約が表示されます。

## GDB コマンド

対応する GDB コマンドは ‘load’ です。

例

注: 個々のメッセージは 1 行に表示されます。ここでは、ページの中に収まるようにメッセージは改行されています。

```
(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)
```

`-target-exec-status` コマンド

#### 概要

`-target-exec-status`

ターゲットの状態に関する情報（例えば、実行中か否かなど）を提供します

#### GDB コマンド

対応する GDB コマンドは存在しません。

#### 例

N.A.

`-target-list-available-targets` コマンド

#### 概要

`-target-list-available-targets`

接続可能なターゲットの一覧を示します。

#### GDB コマンド

対応する GDB コマンドは ‘help target’ です。

#### 例

N.A.

`-target-list-current-targets` コマンド

#### 概要

`-target-list-current-targets`

カレントなターゲットを示します。

#### GDB コマンド

対応する情報が（他の情報とともに） ‘info file’ によって表示されます。

#### 例

N.A.

`-target-list-parameters` コマンド

## 概要

`-target-list-parameters`

## GDB コマンド

対応するコマンドは存在しません。

## 例

N.A.

`-target-select` コマンド

## 概要

`-target-select type parameters ...`

GDB をリモート・ターゲットに接続します。このコマンドは 2 つの引数を取ります。

‘*type*’            ターゲットのタイプ。例えば、‘*async*’、‘*remote*’など。

‘*parameters*’        デバイス名、ホスト名、あるいは、これらに類する情報。詳細については、セクション 13.2 [ターゲットを管理するコマンド], ページ 115 を参照してください。

接続通知が出力された後に、ターゲット・プログラムのアドレスが以下の形式で出力されます。

```
^connected,addr="address",func="function name",
  args={arg list}
```

## GDB コマンド

対応する GDB コマンドは ‘*target*’ です。

## 例

```
(gdb)
-target-select async /dev/ttya
^connected,addr="0xfe00a300",func="??",args={}
(gdb)
```

## 19.12 GDB/MI スレッド・コマンド

`-thread-info` コマンド

## 概要

`-thread-info`

## GDB コマンド

対応する GDB コマンドは存在しません。

例

N.A.

`-thread-list-all-threads` コマンド

概要

`-thread-list-all-threads`

## GDB コマンド

対応する GDB コマンドは `'info threads'` です。

例

N.A.

`-thread-list-ids` コマンド

概要

`-thread-list-ids`

このコマンドの実行時に知られている GDB スレッド ID の一覧を作成します。一覧の末尾には、これらのスレッドの総数も表示されます。

## GDB コマンド

`'info threads'` の提供する情報の一部に同等の情報が含まれています。

例

メイン・プロセス以外にスレッドが存在しない場合の例を以下に示します。

```
(gdb)
-thread-list-ids
^done,thread-ids={},number-of-threads="0"
(gdb)
```

いくつかスレッドが存在する場合の例を以下に示します。

```
(gdb)
-thread-list-ids
^done,thread-ids={thread-id="3",thread-id="2",thread-id="1"},
number-of-threads="3"
(gdb)
```

## -thread-select コマンド

### 概要

`-thread-select threadnum`

*threadnum* により指定される番号を持つスレッドをカレント・スレッドとします。新しくカレント・スレッドとなったスレッドの番号を表示します。また、そのスレッドの最下位フレームを表示します。

## GDB コマンド

対応する GDB コマンドは 'thread' です。

### 例

```

(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\\n\"},
{name="arg",value="0x2"}}},file="vprintf.c",line="31"}
(gdb)

```

## 19.13 GDB/MI トレースポイント・コマンド

トレースポイント・コマンドはまだ実装されていません。

## 19.14 GDB/MI 変数オブジェクト

### GDB/MI において変数オブジェクトを使用する動機

デバッガの変数ウィンドウ ( 局所変数、監視式 (watched expression) など ) の実装について私たちは、Insightによって使われている既存のコードを適合させることを提案しています。

主要な理由は以下の 2 点です。

1. 実績があります ( それは既に第 2 世代に入っています )。

2. 開発に要する時間を短縮してくれます (このことが昨今どれほど重要であるかは改めて言うまでもないでしょう)。

もともとのインターフェイスは Tcl のコードから使うことを想定して設計されています。したがって、flathead を通じて使うことができるようにするために若干の変更が加えられました。このドキュメントでは、利用可能になるであろう flathead オペレーションについて説明し、その使い方に関するヒントを提供します。

注: ここで説明するオペレーション群に加えて、変数ウィンドウの GUI 実装は、少なくとも以下のオペレーションを必要とするであろうと私たちは考えています。

- -gdb-show output-radix
- -stack-list-arguments
- -stack-list-locals
- -stack-select-frame

### GDB/MI における変数オブジェクトの紹介

変数オブジェクトの基本的な考え方は、変数、式、メモリ位置、さらには、CPU レジスタさえも表わす名前付きのオブジェクトを生成することです。生成される個々のオブジェクトにおいて、その属性を調べたり変更したりするためのオペレーション群が利用可能です。

さらに、C の構造体のような複雑なデータ型はツリー・フォーマットによって表わされます。例えば、struct 型の変数がルートとなり、その子ノードが構造体メンバを表わすこととなります。子ノード自体が複雑な型である場合は、その下にまた子ノードが存在することとなります。C、C++、Java に特有の言語上の差異は正しく処理されます。

オブジェクトの実際の値を返す際には、生成される結果において使われる表示フォーマットを個別に選択することが可能です。選択肢は、2 進、10 進、16 進、8 進、および、ナチュラル (natural) です。ナチュラルとは、変数の型に応じて自動的に選択されるデフォルトのフォーマット (int の場合は 10 進、ポインタの場合は 16 進、等) のことです。

以下に、この機能にアクセスするために定義されたすべての flathead オペレーションを示します。

| オペレーション                  | 説明                          |
|--------------------------|-----------------------------|
| -var-create              | 変数オブジェクトを生成する               |
| -var-delete              | 変数オブジェクトとその子を削除する           |
| -var-set-format          | この変数の表示フォーマットを設定する          |
| -var-show-format         | この変数の表示フォーマットを示す            |
| -var-info-num-children   | このオブジェクトが持つ子の数を示す           |
| -var-list-children       | このオブジェクトの子の一覧を返す            |
| -var-info-type           | この変数オブジェクトの型を示す             |
| -var-info-expression     | この変数オブジェクトの表わす対象を表示する       |
| -var-show-attributes     | この変数が書き込み可能であるか、また、存在するかを示す |
| -var-evaluate-expression | この変数の値を取得する                 |
| -var-assign              | この変数の値を設定する                 |
| -var-update              | 変数とその子を更新する                 |

次のサブセクションでは、個々のオペレーションについて詳細に説明し、その使い方を示します。



## 変数オブジェクトに対するオペレーションの説明と用途

`-var-create` コマンド

## 概要

```
-var-create {name | "-"}
           {frame-addr | "*"} expression
```

このオペレーションは、変数、式の評価結果、メモリ・セル、あるいは、CPU レジスタの監視を可能にする変数オブジェクトを生成します。

*name* パラメータは、そのオブジェクトを参照するための文字列です。これは一意でなければなりません。‘-’が指定されると、変数オブジェクトのシステムが自動的に“varNNNNNN” という文字列を生成します。この文字列は、ユーザがこのようなフォーマットの *name* を指定しない限り一意です。名前の重複が検出されるとコマンドの実行は失敗に終わります。

式がその下で評価されるべきフレームは *frame-addr* によって指定することができます。‘\*’は、カレント・フレームが使われるべきことを意味します。

*expression* は、カレントな言語セットにおいて有効な任意の式（ただし、‘\*’以外の文字で始まるものでなければなりません）、あるいは、以下のいずれかです。

- ‘\**addr*’。 *addr* はメモリ・セルのアドレスです。
- ‘\**addr-addr*’ – メモリ・アドレス範囲（検討中）
- ‘\$*regname*’ – CPU レジスタ名

## 結果

このオペレーションは、生成されたオブジェクトの名前、子の数、型を返します。型は、GDB CLI によって作成される文字列のまま返されます。

```
name="name",numchild="N",type="type"
```

`-var-delete` コマンド

## 概要

```
-var-delete name
```

以前に生成された変数オブジェクトとそのすべての子を削除します。

オブジェクト名 *name* を見つけることができない場合、エラーを返します。

`-var-set-format` コマンド

## 概要

```
-var-set-format name format-spec
```

*name* により指定されるオブジェクトの値の出力フォーマットを *format-spec* に設定します。

*format-spec* の構文は以下のとおりです。

```
format-spec ↦
{binary | decimal | hexadecimal | octal | natural}
```

### `-var-show-format` コマンド

#### 概要

`-var-show-format name`

*name* により指定されるオブジェクトの値を表示するのに使われるフォーマットを返します。

*format*  $\mapsto$

*format-spec*

### `-var-info-num-children` コマンド

#### 概要

`-var-info-num-children name`

*name* により指定されるオブジェクトの子の数を返します。

*numchild*=*n*

### `-var-list-children` コマンド

#### 概要

`-var-list-children name`

指定された変数オブジェクトの子の一覧を返します。

*numchild*=*n*, *children*={{*name*=*name*,  
*numchild*=*n*, *type*=*type*}, (repeats N times)}

### `-var-info-type` コマンド

#### 概要

`-var-info-type name`

*name* により指定される変数の型を返します。型は、GDB CLI による出力と同じフォーマットの文字列として返されます。

*type*=*typename*

### `-var-info-expression` コマンド

#### 概要

`-var-info-expression name`

*name* により指定される変数オブジェクトにより表わされる対象を返します。

*lang*=*lang-spec*, *exp*=*expression*

*lang-spec* の構文は{"C" | "C++" | "Java"}で表わされます。

### -var-show-attributes コマンド

#### 概要

`-var-show-attributes name`

`name` により指定される変数オブジェクトの属性の一覧を表示します。

`status=attr [ ( ,attr ) * ]`

`attr` の構文は { { `editable` | `noneditable` } | その他検討中 } で表わされます。

### -var-evaluate-expression コマンド

#### 概要

`-var-evaluate-expression name`

指定された変数オブジェクトによって表わされる式を評価して、その値を文字列として返します。文字列のフォーマットは、そのオブジェクトに対して指定されているカレントなフォーマットです。

`value=value`

### -var-assign コマンド

#### 概要

`-var-assign name expression`

`expression` により指定される式の値を、`name` により指定される変数オブジェクトに代入します。そのオブジェクトは “`editable`” (書き込み可能) でなければなりません。

### -var-update コマンド

#### 概要

`-var-update {name | "*"}`

`name` により指定される変数オブジェクトの値を、その式を構成する値をメモリまたはレジスタから新たに取り直して式を再評価した後に、更新します。‘\*’を指定すると、既存のすべての変数オブジェクトが更新されることになります。

## 19.15 GDB/MI 出力構文の変更案

既存の GDB/MI 出力構文において認められていた問題点の 1 つに、個々の値が一意なラベルを持つ以下のような組

```
{number="1",type="breakpoint",disp="keep",enabled="y"}
```

と、以下のような

```
{ "1", "2", "4" }
{ bp="1", bp="2", bp="4" }
```

値がラベルを持たないリストやラベルが重複しているリストを区別することが困難であるということがあります。

以下に示すのは、この問題を解決するための出力仕様改訂案です。

GDB/MI からの出力は、ゼロ個以上の帯域外レコードから構成されます。場合によってはこの後に、最後に入力されたコマンドに対応する結果レコードが 1 個続くことがあります。この出力シーケンスの終端は、“(gdb)” という文字列によって示されます。

非同期の GDB/MI 出力もこれに類似しています。

ある入力コマンドに直接関連付けられている出力レコードには、その入力コマンドの *token* が接頭語として付与されます。

```
output ↦ { out-of-band-record } [ result-record ] "(gdb)" nl
result-record ↦
    [ token ] "^" result-class { "," result } nl
out-of-band-record ↦
    async-record | stream-record
async-record ↦
    exec-async-output | status-async-output | notify-async-output
exec-async-output ↦
    [ token ] "*" async-output
status-async-output ↦
    [ token ] "+" async-output
notify-async-output ↦
    [ token ] "=" async-output
async-output ↦
    async-class { "," result } nl
result-class ↦
    "done" | "running" | "connected" | "error" | "exit"
async-class ↦
    "stopped" | 開発途中のためこれ以外のものは必要に応じて追加予定
result ↦ string "=" value
value ↦ c-string | tuple | list
tuple ↦ "{ }" | "{ result { "," result } }"
list ↦ "[ ]" | "[ value { "," value } ]"
string ↦ [-A-Za-z\.\0-9_]*
c-string ↦
    入力仕様を参照
stream-record ↦
    console-stream-output | target-stream-output | log-stream-output
```

*console-stream-output*  $\mapsto$   
"~" *c-string*

*target-stream-output*  $\mapsto$   
"@ " *c-string*

*log-stream-output*  $\mapsto$   
"&" *c-string*

*nl*  $\mapsto$  CR | CR-LF

*token*  $\mapsto$  "連続した任意の数字"

さらに、以下のものが現在開発中です。

*query* このアクションは現在のところまだ未定義です。

注:

- すべての出力シーケンスは、ピリオドを含む単一行によって終了します。
- *token*は対応するリクエストから取られます。実行コマンドが-exec-interrupt コマンドによって割り込まれる場合、'\*stopped' メッセージに関連付けられるトークンは、interrupt-command のものではなく、もともとの実行コマンドのものです。
- *status-async-output* には、時間のかかるオペレーションの進行具合に関する現在のステータス情報が含まれます。これは破棄することもできます。すべてのステータス出力には、接頭語として '+' が付きます。
- *exec-async-output* には、ターゲットの非同期的な状態変化（実行停止、実行開始、消滅）に関する情報が含まれます。すべての非同期出力には、接頭語として '\*' が付きます。
- *notify-async-output* には、クライアントが対処すべき補足情報（例えば、新たなブレークポイント情報）が含まれます。すべての通知出力には、接頭語として '=' が付きます。
- *console-stream-output* は、コンソールにおいてそのまま表示されるべき出力です。それは、CLI コマンドに対するテキストによる応答です。すべてのコンソール出力には、接頭語として '~' が付きます。
- *target-stream-output* は、ターゲット・プログラムにより生成される出力です。すべてのターゲット出力には、接頭語として '@' が付きます。
- *log-stream-output* は、GDB 内部からの出力テキストです。例えば、エラー・ログの一部として表示されるべきメッセージです。すべてのログ出力には、接頭語として '&' が付きます。



## 20 GDB のバグ報告

ユーザからのバグ報告は、GDB の信頼性を向上させるのに重要な役割を果たしています。

バグを報告することで、その問題の解決につながり、結果として報告者自ら利益を得ることができるかもしれません。もちろん、何の解決にもつながらないこともあります。しかし、いずれにしても、バグ報告の主要な意義は、次のバージョンの GDB をより良いものにすることで、コミュニティ全体の役に立つという点にあります。バグ報告は、GDB の保守作業へのユーザからの貢献です。

バグ報告がその目的とするところを首尾よく達成できるようにするためには、バグを修正することを可能にするような情報が提供されなければなりません。

### 20.1 本当にバグを見つけたのかどうかを知る方法

発見した現象がバグかどうかよく分からない場合には、以下のガイドラインを参照してください。

- 入力された情報が何であれ、デバグが致命的なシグナルを受信するのであれば、それは GDB のバグです。信頼性のあるデバグは決してクラッシュなどしません。
- 正当な入力に対して GDB がエラー・メッセージを出力するのであれば、それはバグです。(クロス・デバグを行っている場合には、ターゲットへの接続に問題がある可能性もあるということに注意してください。)
- 不正な入力に対して GDB がエラー・メッセージを出力しないのであれば、それはバグです。ただし、ユーザにとって「不正な入力」に思えるものが、実は「拡張機能」であったり「古くから使われている用法のサポート」であったりすることもあります。
- デバグ・ツールに関する経験が豊富なユーザからの GDB の改善提案は、どのような場合でも歓迎です。<sup>1</sup>

### 20.2 バグの報告方法

いくつかの企業や個人が GNU のソフトウェアをサポートしています。こうしたサポート組織から GDB を入手したのであれば、まずその組織に連絡することをお勧めします。

サポートを提供している多くの企業、個人の連絡先情報が、GNU Emacs ディストリビューションの 'etc/SERVICE' ファイルに記載されています。

どのような場合でも、GDB のバグ報告を (英語で) 以下のアドレスに送ることをお勧めします。<sup>2</sup>

`bug-gdb@gnu.org`

'info-gdb'、'help-gdb'、および、いかなるニュースグループにもバグ報告を送ることはしないでください。GDB ユーザのほとんどは、バグ報告を受け取りたいと考えてはいません。バグ報告を受け取りたいと思っている人は、'bug-gdb' の配信を受けるようにしているはずです。

メーリング・リスト 'bug-gdb' には、リピータとして機能する 'gnu.gdb.bug' というニュースグループがあります。このメーリング・リストとニュースグループは、全く同一のメッセージを配信しています。メーリング・リストではなくニュースグループにバグ報告を流そうと考える人

<sup>1</sup> 訳注: この日本語の翻訳マニュアルへの改善提案は、`ki@home.email.ne.jp` に送ってください。

<sup>2</sup> 訳注: この日本語の翻訳マニュアルのバグは、日本語 (か英語) で、`ki@home.email.ne.jp` に報告してください。

がよくいます。これはうまく機能するように見えますが、1つ重大な問題があります。ニュースグループへの投稿では、送信者へのメール・パスが分からないことがよくあります。したがって、もっと多くの情報が必要になったときに、バグの報告者と連絡を取ることができない可能性があります。こういふことがあるので、メーリング・リストへのバグ報告の方が望ましいのです。最後の手段として、バグ報告を（英語で）紙に書いて下記に郵送するという方法があります。

GNU Debugger Bugs  
Free Software Foundation Inc.  
59 Temple Place - Suite 330  
Boston, MA 02111-1307  
USA

役に立つバグ報告を行うための最も根本的な原則は、すべての事実を報告することです。ある事実を書くべきか省くべきかよく分からない場合は、書くようにしてください。

事実が省略されてしまうことがよくありますが、これはバグ報告者が、自分には問題の原因は既に分かっていると考え、いくつかの細かい点は関係がないと仮定してしまうからです。したがって、例の中で使った変数の名前などは重要ではないと、報告者は考えます。おそらくそうかもしれませんが、しかし、完全にそうであるとも言い切れません。メモリの参照がデタラメな場所を指しているというバグで、それがたまたまメモリ上においてその名前が置かれている箇所から値を取り出しているということがあるかもしれません。名前が異なれば、その内容は、バグが存在するにもかかわらずデバッガが正しく動作してしまうような値になるかもしれません。このようなことがないように、特定の完全な実例を提供してください。バグの報告者にとっては、このようにするのが最も簡単なはずであり、かつ、それが最も役に立つのです。

バグ報告の目的は、そのバグを修正することができるようにすることにある、という点を頭に入れておいてください。そのバグが、以前に報告されたものと同じであるという可能性もありますが、バグ報告が完全なもので、必要な情報がすべて含まれたものでなければ、バグの報告者にも私たちにもそのことを知ることはできません。

ときどき、2、3の大雑把な事実だけを記述して、「何か思い当たることはありますか?」と聞いてくる人がいます。このようなバグ報告は役に立ちません。このような報告には、より適切なバグ報告を送るよう報告者に注意する場合を除いて、返事することを拒否するよう強くお願いします。

バグを修正できるようにするためには、報告者は以下の情報をすべて含めるべきです。

- GDB のバージョン。GDB のバージョンは、引数を指定せずに GDB を起動すると、表示されます。また、いつでも `show version` コマンドで表示させることができます。この情報がないと、カレント・バージョンの GDB を使ってバグを探すことに意味があるかどうかを知ることができません。
- 使っているマシンのタイプ、オペレーティング・システムの名前とバージョン番号。
- GDB をコンパイルするのに使われたコンパイラ（および、そのバージョン）。例えば、`gcc-2.8.1`。
- デバッグ対象のプログラムをコンパイルするのに使われたコンパイラ（および、そのバージョン）。例えば、`gcc-2.8.1`、あるいは、`HP92453-01 A.10.32.03 HP C コンパイラ`。GCC については、`gcc --version` によってこの情報を知ることができます。他のコンパイラについては、そのドキュメントを参照してください。
- バグを見つけたプログラムをコンパイルする際に、コンパイラに渡したコマンド引数。例えば、`'-O'` オプションを使ったか否かなど。何か重要な点を省いてしまうことがないように、すべての引数を記述してください。‘`Makefile`’のコピー（あるいは、`make`からの出力）を添付すれば十分でしょう。



引数が何であったのかを私たちが推測しようとしても、おそらく誤った推測をしてしまうでしょう。そうすると、バグは再現しないかもしれません。

- バグを再現することのできる、完全な入力スクリプトとすべての必要なソース・ファイル。
- 発見された、正しくないと思われる動作の説明。例えば、「致命的なシグナルを受信する」など。

もちろん、GDB が致命的なシグナルを受信するというバグであれば、私たちも間違いなくそれに気がつくでしょう。しかし、出力が正しくないというバグであれば、紛れもない誤りでなければ、私たちはそれに気付かないかもしれません。私たちが間違いをする可能性を排除するようにしてください。

たとえ致命的なシグナルを受信するような問題であっても、報告者はそのことを明示的に報告すべきです。何か奇妙なことが起こっていると仮定しましょう。例えば、報告者が使っている GDB にちぐはぐなところがあるとか、報告者のシステム上にある C ライブラリのバグだった、というような場合です（こういうことは、実際にありました！）。このような場合、報告者の GDB はクラッシュしても、私たちのところではクラッシュしません。クラッシュするはずであると報告されていれば、私たちの GDB がクラッシュしなくても、「私たちのところではバグが発生しない」ということを知ることができます。クラッシュするはずであるという報告がなければ、実際の現象から何も結論を引き出すことができません。

- もし GDB のソースへの修正を提案したいのであれば、コンテキスト付きの差分情報を送ってください。GDB のソースについて何か議論する場合も、行番号に言及するのではなく、コンテキストに言及してください。

私たちが開発中のソースの行番号は、報告者の持っているソースの行番号とは一致しないでしょう。報告者から見たソースの行番号は、私たちにとって役に立つ情報を提供してくれません。

以下に、バグ報告に必要なではない情報をいくつか列挙します。

- バグの包括的な説明。

バグを見つけると、多くの時間をかけて、入力ファイルをどのように変更するとバグが発生しなくなり、どのように変更した場合はバグが発生し続けるかを調べる人がよくいます。

これは多くの場合、時間のかかる作業であり、しかもあまり役に立ちません。というのは、私たちがバグを見つけるのは、デバッガでブレイクポイントを使いながら 1 つの実例を実行させることによってであり、一連の実例からの純粋な演繹によってではないからです。時間を無駄にせず、何かほかのことに使うようお勧めします。

もちろん、一番最初にバグを見つけたときの実例の代わりとなる、もっと単純な実例を見つけることができるのであれば、私たちにとっても便利です。出力におけるエラーはより発見しやすいものですし、デバッガ配下で実行させる方が時間がかかりません。

しかし、単純化は絶対に必要というわけでもありません。こういうことをしたくないのであれば、バグを発見したときのテスト・ケース全体を送って、バグの報告を行ってください。

- バグに対するパッチ。

バグに対するパッチは、それが良いものであれば、役に立ちます。しかし、パッチがあれば十分であるとみなして、テスト・ケースのような必要な情報を送るのを省かないでください。提供されたパッチに問題があり、別の方法で問題を修正することにする場合もありますし、提供されたパッチを全く理解できないということもあるかもしれません。

GDB のような複雑なプログラムでは、コード中のある特定のパスを通るような実例を作成するのは困難なことがあります。報告者が実例を送ってくれないければ、私たちには実例を作

成することができず、したがって、バグが修正されたことを検証することができなくなってしまいます。

また、報告者の送ってくれたパッチがどのような問題を修正しようとしているのか私たちに理解できない場合、あるいは、なぜそのパッチが改善になるのか私たちが理解できない場合、そのパッチを組み込むことはしません。テスト・ケースが1つでもあれば、そうしたことを理解するのに役立つでしょう。

- バグが何であるか、あるいは、何に依存しているかに関する推測。  
このような推測は普通は間違っているものです。私たちですら、デバッガを使って事実を見出すまでは、このような点に関して正しく推測することはできないのです。

## 21 コマンドライン編集

この章では、GNU のコマンドライン編集インターフェイスの基本的な特徴について説明します。

### 21.1 行編集入門

以下のパラグラフでは、キー・ストロークを表わすために使用される表記法について説明します。

**C-k** は、Control-K という意味です。これは、コントロール・キーが押されたままの状態でも **k** が押されたときに生成される文字を表わします。

**M-k** は、Meta-K という意味です。これは、メタ・キー（があるものとして、それ）が押されたままの状態でも **k** が押されたときに生成される文字を表わします。メタ・キーがない場合、最初に **ESC** キーを押し、次に **k** を押すことで、同等のキー・ストロークを生成することができます。どちらの手順も、**k** をメタ化する、といえます。

**M-C-k** は、Meta-Control-K という意味です。これは、**C-k** をメタ化することにより生成される文字を指します。

さらに、いくつかのキーには名前があります。**DEL**、**ESC**、**LFD**、**SPC**、**RET**、**TAB** は、この文章の中でも、初期化ファイルの中でも、各々のキーを表わします（セクション 21.3 [Readline Init File], ページ 244 参照）。

### 21.2 Readline の操作

対話的なセッションにおいて、長いテキストを 1 行に記述した後で、その行の先頭の単語のスペルが間違っていたことに気が付くことがよくあります。Readline ライブラリは、入力したテキストを操作するための一連のコマンドを提供しており、これによって、その行の大部分を入力し直すことなく、タイプ・ミスしたところだけを修正することができます。これらの編集コマンドを使って、修正が必要なところにカーソルを移動させ、テキストを削除したり、修正テキストを挿入したりします。その行の修正が終われば、単に **RET** を押します。**RET** を押すのに、行末にいる必要はありません。カーソルが行内のどこにあると、その行全体が入力として受け付けられます。

#### 21.2.1 Readline の基本

行内に文字を入力するには、単にその文字をタイプします。タイプされた文字はカーソルの位置に表示され、カーソルは 1 桁分右へ移動します。1 文字打ち間違えた場合は、削除文字（erase character）を使って、後退しながら打ち間違えた文字を削除することができます。

ときには、本当は入力したかった文字を入力せず、その誤りに気が付くことなく、さらに数文字を入力してしまうことがあります。このような場合には、**C-b** によってカーソルを左に移動し、誤りを訂正することができます。訂正後、**C-f** によってカーソルを右に移動することができます。

行の途中でテキストを追加すると、挿入されたテキストのためのスペースを空けるために、カーソルの右側にある文字が右方向に押しやられることに気がつくでしょう。同様に、カーソル位置にあるテキストを削除すると、テキストが削除されたために生じる空白を埋めるために、カーソルの右側にある文字が左方向に引き戻されます。入力行のテキストを編集するための最も基本的な操作の一覧を以下に示します。

**C-b**      1 文字戻ります。

- `C-f` 1 文字進みます。
- `DEL` カーソルの左にある文字を削除します。
- `C-d` カーソル位置にある文字を削除します。
- 表示可能な文字  
行内のカーソル位置にその文字を挿入します。
- `C-^` 最後の編集コマンドを取り消して元に戻します。行内に文字が無くなるまで取り消しを繰り返すことが可能です。

### 21.2.2 Readline 移動コマンド

上記の一覧は、ユーザが入力行を編集するのに必要な、最も基本的なキー・ストロークを説明したものです。ユーザの利便を考慮して、`C-b`、`C-f`、`C-d`、`DEL`に加えて多くのコマンドが追加されてきました。以下に、行内をより迅速に動きまわるためのコマンドをいくつか示します。

- `C-a` 行の先頭に移動します。
- `C-e` 行の末尾に移動します。
- `M-f` 1 単語分先に進みます。単語は、文字と数字から構成されます。
- `M-b` 1 単語分前に戻ります。
- `C-l` 画面上の情報を消去し、カレント行が画面の一番上にくるようにして再表示します。

`C-f`が1文字分先に進むのに対して、`M-f`が1単語分先に進む点に注意してください。大まかな慣例として、コントロール・キーを使うと文字単位の操作になり、メタ・キーを使うと単語単位の操作になります。

### 21.2.3 Readline キル (kill) コマンド

テキストをキル (kill) するとは、行からテキストを削除し、その際に、そのテキストを後に引き出して行内にヤंक (yank)<sup>1</sup> することができるように退避しておくことを指します。あるコマンドの説明に「テキストをキルする」という記述があれば、後に別の箇所 (あるいは同じ箇所) において、そのテキストを再入手することができると考えて間違いありません。

キル・コマンドを使うと、テキストはキル・リング (kill-ring) に退避されます。キル・コマンドを任意の回数連続して実行すると、キルされたテキストはすべて連結されて退避されます。したがって、ヤंकを行うと、そのすべてを入手することができます。キル・リングは個々の行に固有のものではありません。以前入力した行においてキルしたテキストを、後になって別の行を入力しているときにヤंकすることができます。

以下に、テキストをキルするためのコマンドを一覧で示します。

- `C-k` カレントなカーソル位置から行末までのテキストをキルします。
- `M-d` カーソル位置から、カーソルの置かれている単語の末尾までをキルします。カーソルが2つの単語の間にあるときは、次の単語の末尾までをキルします。
- `M-DEL` カーソル位置から、カーソルの置かれている単語の先頭までをキルします。カーソルが2つの単語の間にあるときは、前の単語の先頭までをキルします。

<sup>1</sup> 訳注：最後にキルされたテキストを再挿入すること

**(C-w)** カーソル位置から、それより前にある最初の空白類までをキルします。単語間の境界が異なるので、これは **(M-DEL)** とは異なります。

キルされたテキストを引き出して行内へヤंकする方法を、以下に示します。ヤंकとは、最後にキルされたテキストを、キル・バッファからコピーすることを意味しています。

**(C-y)** バッファ内のカーソル位置に、最後にキルされたテキストをヤंकします。

**(M-y)** キル・リングを回転させ、新たに一番上にきたテキストをヤंकします。このコマンドを実行できるのは、1 つ前に実行したコマンドが **(C-y)** または **(M-y)** の場合だけです。

### 21.2.4 Readline の引数

Readline コマンドには数値引数を渡すことができます。数値引数は、繰り返し回数として使われたり、引数の符号として使われたりします。通常は先に進むようなコマンドに負の数を引数として指定すると、前に戻るようになります。例えば、行の先頭までのテキストをキルするには、`'M-- C-k'` としてもよいでしょう。

コマンドに数値引数を渡す通常の方法は、コマンドの前にメタ化された数字を入力することです。入力された最初の「数字」がマイナス記号 ( `-` ) の場合、引数の符号は負になります。引数を開始するためには、メタ化された数字を 1 つだけ入力すればよく、残りの数字はそのまま入力することができます。そして最後にコマンドを入力します。例えば、**(C-d)** コマンドに引数として 10 を渡すためには、`'M-1 0 C-d'` と入力します。

### 21.2.5 履歴中のコマンドの検索

readline は、コマンド履歴の中から、指定された文字列を含む行を検索するコマンドを提供しています。インクリメンタル ( *incremental* ) と非インクリメンタル ( *non-incremental* ) の 2 つの検索モードがあります。

インクリメンタル ( *incremental* ) ・モードでは、ユーザが検索文字列を入力し終わる前から検索が始まります。検索文字列の中の文字が 1 つ入力されるたびに、Readline は、それまで入力された文字列にマッチする、履歴の中の次のエントリを表示します。インクリメンタル・モードの検索では、検索したい履歴エントリを見つけるのに本当に必要となる文字だけを入力するだけで済みます。インクリメンタル・モードの検索を中止するのには、`isearch-terminators` 変数の値の中に含まれる文字が使われます。この変数に値が割り当てられていない場合は、**(ESC)** 文字や **(C-j)** によってインクリメンタル・モードの検索が中止されます。**(C-g)** は、インクリメンタル・モードの検索を終了させて、元の行を表示します。検索が中止されると、検索文字列を含む履歴エントリがカレント行となります。検索文字列にマッチする他のエントリを履歴リストからを見つけるためには、必要に応じて **(C-s)** または **(C-r)** を入力します。これによって、それまでに入力された検索文字列にマッチする次のエントリを履歴からを見つけるために、下の方向、または、上の方向に検索が行われます。Readline コマンドにバインドされているキー・シーケンスのうち上記以外のものを入力すると、検索は中止され、そのコマンドが実行されます。例えば **(RET)** が入力されると、検索は中止され、そのときの行が受け入れられたことになります。したがって、履歴リストの中のそのコマンドが実行されます。

非インクリメンタル ( *non-incremental* ) ・モードでは、マッチする履歴行の検索を開始する前に、検索文字列全体を読み込みます。検索文字列は、ユーザによって入力されたものでも構いませんし、カレント行の内容の一部であっても構いません。

## 21.3 Readline 初期化ファイル

Readline ライブラリには、emacsスタイルのキー・バインディングがデフォルトで組み込まれていますが、異なるキー・バインディングを使うこともできます。ホーム・ディレクトリ内のファイル *inputrc* にコマンドを記述することで、誰でも Readline を使うプログラムをカスタマイズすることができます。このファイルの名前は、環境変数 INPUTRC の値から取られます。この変数に値がセットされていない場合のデフォルトは、`~/inputrc` です。

Readline ライブラリを使うプログラムが起動されると、初期化ファイルが読み込まれ、キー・バインディングが設定されます。

さらに、`C-x C-r` コマンドを実行すると、この初期化ファイルが再読み込みされます。初期化ファイルに変更が加えられていれば、その変更が反映されます。

### 21.3.1 Readline 初期化ファイルの構文

Readline 初期化ファイルの中では、ほんの少数の基本的な構文だけが使用できます。空行は無視されます。'#' で始まる行はコメントです。'\$' で始まる行は、条件構文を表わします (セクション 21.3.2 [Conditional Init Constructs], ページ 248 参照)。その他の行は、変数設定とキー・バインディングを示します。

**変数設定**     初期化ファイルの中で `set` コマンドを使用して Readline の変数の値を変更することによって、Readline の実行時の振る舞いを変更することができます。デフォルトの Emacs スタイルのキー・バインディングを変更して、vi の行編集コマンドを使用できるようにするには、以下のようにします。

```
set editing-mode vi
```

以下の変数によって、実行時の振る舞いのかなりの部分が変更可能です。

**bell-style**

Readline が端末のベル音を鳴らしたいと判断した場合に、何が起るかを制御します。'none' がセットされると、Readline はベル音を鳴らしません。'visible' がセットされると、視覚的なベル<sup>2</sup> が利用可能であれば、それを使います。'audible' (デフォルト) がセットされると、Readline は、端末のベル音を鳴らそうと試みます。

**comment-begin**

`insert-comment` コマンドが実行されたときに、行の先頭に挿入される文字列です。デフォルトの値は '#' です。

**completion-ignore-case**

'on' がセットされると、Readline は、大文字・小文字を区別せずに、ファイル名のマッチングや補完を行います。デフォルトの値は 'off' です。

**completion-query-items**

ユーザに対して補完候補の一覧を見たいかどうか問い合わせるタイミングを決定する、補完候補の数です。補完候補の数がこの値よりも多いと、Readline は、補完候補の一覧を見たいかどうかをユーザに対して問い合わせることになります。この値よりも少ない場合は、問い合わせを行うことなく一覧を表示します。デフォルトの境界は 100 です。

<sup>2</sup> 訳注：ベル音を鳴らす代わりに、画面表示をフラッシュさせることを表わしています。

**convert-meta**

‘on’がセットされると、Readlineは、第8ビットがセットされている文字をASCIIのキー・シーケンスに変換します。これは、該当文字の第8ビットを落として、その前に`(ESC)`文字を付加することで、メタ・プレフィックス・キー・シーケンス ( meta-prefixed key sequence ) に変換することによって行われます。デフォルトの値は‘on’です。

**disable-completion**

‘On’がセットされると、Readlineは単語補完を抑制します。補完文字 ( completion character ) は、あたかも self-insert にマップされたかのように、行内に挿入されます。デフォルトは‘off’です。

**editing-mode**

editing-mode変数は、デフォルトで使用するキー・バインディングの種類を制御します。Readlineは、デフォルトの状態では、Emacs編集モードで起動します。このモードは、キー・ストロークがEmacsに非常に良く似ています。この変数は、‘emacs’と‘vi’のどちらかに設定することができます。

**enable-keypad**

‘on’がセットされると、Readlineは、呼び出されたときに、アプリケーション・キーパッド ( application keypad ) を有効にすることを試みます。システムによっては、矢印キーを使用できるようにするために、これが必要となります。デフォルトは‘off’です。

**expand-tilde**

‘on’がセットされると、Readlineが単語補完を試みる際に、チルダの展開が行われます。デフォルトは‘off’です。

**horizontal-scroll-mode**

この変数は、‘on’と‘off’のどちらかに設定することができます。これを‘on’に設定すると、1行のテキストの長さがスクリーン幅よりも長い場合に、編集行のテキストが次の行に折り返すことなく、同じ行の上で水平方向にスクロールするようになります。デフォルトでは、この変数には‘off’がセットされています。

**input-meta**

‘on’がセットされると、Readlineは、8ビット入力に対する端末側のサポートがどうであれ、8ビット入力を有効にします ( 読み取られた文字の第8ビットを落としません )。デフォルト値は‘off’です。meta-flagは、この変数の別名です。

**isearch-terminators**

インクリメンタル・モードの検索を停止させるべき文字の集合。これらの文字は、それまでに入力されてきた検索コマンドの一部として使われることはありません ( セクション 21.2.5 [履歴中のコマンドの検索], ページ 243 参照 )。この変数に値が割り当てられていない場合、`(ESC)`と`(C-J)`が、インクリメンタル・モードの検索を停止させる文字となります。

**keymap**

Readlineが認識している、キー・バインディング・コマンドのカレントなキーマップをセットします。セットすることのできるkeymap名は、

emacs、emacs-standard、emacs-meta、emacs-ctlx、vi、vi-command、vi-insertです。viは vi-commandと同等です。また、emacsは emacs-standardと同等です。デフォルトの値は、emacsです。editing-mode変数の値も、デフォルトのキーマップに影響を及ぼします。

#### mark-directories

‘on’がセットされると、補完されたディレクトリ名の後ろにスラッシュが付加されます。デフォルトは‘on’です。

#### mark-modified-lines

この変数に‘on’がセットされると、Readlineは、変更された履歴行の先頭にアスタリスク(‘\*’)を表示します。この変数は、デフォルトでは‘off’です。

#### output-meta

‘on’がセットされると、Readlineは、第8ビットがセットされている文字を、メタ・プレフィックス・エスケープ・シーケンス( meta-prefixed escape sequence )としてではなく、直接表示します。デフォルトは‘off’です。

#### print-completions-horizontally

‘on’がセットされると、Readlineは、マッチする補完候補をアルファベット順にソートして、画面の下向きではなく、水平方向に並べて表示します。デフォルトは‘off’です。

#### show-all-if-ambiguous

補完関数のデフォルトの振る舞いを変更します。‘on’がセットされると、複数の補完候補を持つ単語は、ベル音を鳴らすことなく、直ちに補完候補を一覧表示させます。デフォルト値は‘off’です。

#### visible-stats

‘on’がセットされると、補完候補を一覧表示する際に、ファイル・タイプを示す文字がファイル名の後ろに付加されます。デフォルトは‘off’です。

### キー・バインディング

初期化ファイルの中でキー・バインディングを制御するための構文は単純です。まず、キー・バインディングを変更したいコマンドの名前を知っている必要があります。以下のセクションにおいて、コマンドの名前、そのコマンドにデフォルトのキー・バインディングがある場合はそのバインディング、および、そのコマンドが何をやるものであるかについての簡単な説明を、一覧にして示します。

コマンドの名前を知っていれば、初期化ファイルの中で、コマンドにバインドしたいキーの名前、コロン、そして最後にコマンドの名前を、1行にして記述するだけです。キーの名前は、好みに応じて異なる方法で表現することができます。

keyname: function-name または macro

keyname は、英語で記述されたキーの名前です。例えば、以下のようになります。

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
```



```
Control-o: "> output"
```

上の例では、`C-u` が関数 `universal-argument` にバインドされ、`C-o` がその右側に記述されたマクロ（行内に `> output` というテキストを挿入するマクロ）を実行するようバインドされます。

"keyseq": *function-name* または *macro*

前の例の *keyname* とは異なり、*keyseq* には、キー・シーケンス全体を示す文字列を指定することができます。これは、キー・シーケンスを二重引用符で囲むことによって実現されます。以下の例に示すように、いくつかの GNU Emacs スタイルのキー・エスケープを使うことができますが、特殊文字の名前は認識されません。

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

上の例では、`C-u` が（最初の例と同様）関数 `universal-argument` に、`C-x C-r` が関数 `re-read-init-file` に、`ESC` `[` `11` `~` が `'Function Key 1'` というテキストを挿入するよう、それぞれバインドされています。

キー・シーケンスを指定する際には、以下の GNU Emacs スタイルのエスケープ・シーケンスが利用できます。

|                  |                |
|------------------|----------------|
| <code>\C-</code> | コントロール・プレフィックス |
| <code>\M-</code> | メタ・プレフィックス     |
| <code>\e</code>  | エスケープ文字        |
| <code>\\</code>  | バックスラッシュ       |
| <code>\"</code>  | <code>"</code> |
| <code>\'</code>  | <code>'</code> |

GNU Emacs スタイルのエスケープ・シーケンスに加えて、別のバックスラッシュ・エスケープ群が利用できます。

|                    |                                                       |
|--------------------|-------------------------------------------------------|
| <code>\a</code>    | 警告（ベル）                                                |
| <code>\b</code>    | バックスペース                                               |
| <code>\d</code>    | 削除                                                    |
| <code>\f</code>    | フォーム・フィード                                             |
| <code>\n</code>    | 改行                                                    |
| <code>\r</code>    | 復帰（carriage return）                                   |
| <code>\t</code>    | 水平タブ                                                  |
| <code>\v</code>    | 垂直タブ                                                  |
| <code>\nnn</code>  | ASCII コードが 8 進数値の <i>nnn</i> （1 個以上 3 個以下の数字）に相当する文字  |
| <code>\xnnn</code> | ASCII コードが 16 進数値の <i>nnn</i> （1 個以上 3 個以下の数字）に相当する文字 |

マクロのテキストを入力する際には、マクロ定義であることを示すために、単一引用符または二重引用符を使わなければなりません。引用符に囲まれないテキストは、関数名であると見なされます。マクロ本体においては、上記のバックスラッシュ・エスケープは展開されます。バックスラッシュとそれに続く文字の組み合わせがバックスラッシュ・エスケープに該当しない場合、マクロのテキストの中のバックスラッシュは、`'\"'`や`'\"'`も含めて、直後にある文字を引用します。例えば、以下のバインディングによって、`'C-x \"'`は、行内に`'\"'`を1つ挿入することになります。

```
"\C-x \"": "\"\""
```

### 21.3.2 条件初期化構文

Readline は、C のプリプロセッサにおける条件コンパイル機能と質的に類似した機能を実装しています。これによって、あるテストの結果に応じてキー・バインディングや変数設定が実行されるようにすることができます。4 種類のパーサ指示子が使われます。

**\$if**            \$if は、編集モード、使用されている端末、あるいは、Readline を使用しているアプリケーションに応じてバインディングが行われるようにすることを可能にします。  
\$if の後ろに、テストされる内容が行末まで続きます。テストされる内容をほかのものと分離するために特別に文字を使う必要はありません。

**mode**            Readline が emacs モードと vi モードのどちらで動作しているかをテストするために、\$if 指示子の一形式である `mode=` が使用されます。例えば、Readline が emacs モードで開始されている場合のみ、`emacs-standard` や `emacs-ctlx` のキーマップでバインディングをセットするようにするために、これを `'set keymap'` コマンドと組み合わせて使用することができます。

**term**            `term=` という形式は、端末のファンクション・キーによって特定のキー・シーケンスが出力されるようなバインディングを行うなどの目的で、端末固有のキー・バインディングを組み込むために使用することができます。`'='` の右側の単語は、端末の完全名と、端末の名前のうち最初の`'-'`までの部分の両方に対してテストされます。これにより、例えば `sun` は、`sun` と `sun-cmd` の両方にマッチすることになります。

**application**    *application* は、アプリケーション固有の設定を組み込むために使用されます。Readline ライブラリを使用する個々のプログラムがセットする *application name* (アプリケーション名) をテストすることができます。特定のプログラムにとって役に立つ関数に対してキー・シーケンスをバインドするために、これを使用することができます。例えば以下のコマンドは、Bash において、カレントな単語、または、1 つ前の単語を引用符で囲むキー・シーケンスを追加します。

```
$if Bash
# Quote the current or previous word
"\C-xq": "\eb\""\ef\"""
$endif
```

**\$endif**            このコマンドは、前の例が示すように、\$if コマンドを終わらせます。

**\$else**            \$if 指示子から枝分かれしたこの部分に記述されたコマンドは、テスト結果が偽であった場合に実行されます。

`$include` この指示子は、引数としてファイル名を 1 つ取り、そのファイルからコマンドとバインディングを読み込みます。

```
$include /etc/inputrc
```

### 21.3.3 初期化ファイルのサンプル

以下に、`inputrc` ファイルの実例を示します。この中では、キー・バインディング、変数割り当て、条件構文の例が示されています。

```

# このファイルは、Gnu Readline ライブラリを使うプログラムの行入力編集
# の振る舞いを制御する。Gnu Readline ライブラリを使うプログラムには、
# FTP、Bash、Gdb などがある。
#
# inputrc ファイルは、C-x C-r によって再読み込みすることができる。
# '#' で始まる行は、コメントである。
#
# 最初に、/etc/Inputrc からシステム全体のバインディングと変数割り当て
# を取り込む。
$include /etc/Inputrc

#
# emacs モードにおける種々のバインディングをセットする。

set editing-mode emacs

$if mode=emacs

Meta-Control-h:  backward-kill-word  関数名の後ろのテキストは無視される。

#
# キーパッド・モードにおける矢印キー
#
# "\M-OD":      backward-char
# "\M-OC":      forward-char
# "\M-OA":      previous-history
# "\M-OB":      next-history
#
# ANSI モードにおける矢印キー
#
# "\M-[D":      backward-char
# "\M-[C":      forward-char
# "\M-[A":      previous-history
# "\M-[B":      next-history
#
# 8 ビット・キーパッド・モードにおける矢印キー
#
# "\M-\C-OD":   backward-char
# "\M-\C-OC":   forward-char
# "\M-\C-OA":   previous-history
# "\M-\C-OB":   next-history
#
# 8 ビット ANSI モードにおける矢印キー
#
# "\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

# 旧スタイルのバインディング。これがたまたまデフォルトでもある。
TAB: complete

# シェルとのやりとりにおいて便利なマクロ
$if Bash
# パス (PATH) の編集
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# 引用符で囲まれた単語を入力するための準備 -- 先頭と末尾の二重引用符
# を挿入して、先頭の引用符の直後に移動
"\C-x\"": "\""\C-b"
# バックスラッシュを挿入
# (シーケンスやマクロにおいて、バックスラッシュ・エスケープをテストする)
"\C-x\\": "\\\"
# カレントな単語、または、1つ前の単語を引用符で囲む
"\C-xq": "\eb\""\ef\"
# バインドされていない行再表示コマンドにバインディングを追加
"\C-xr": redraw-current-line
# カレント行において変数を編集
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# 視覚的なベルが利用可能であれば、それを使う
set bell-style visible

# 読み込みの際に、文字の第8ビットを落とさない
set input-meta on

# iso-latin1 文字は、プレフィックス・メタ・シーケンスに変換せず、
# そのまま挿入する
set convert-meta off

# 第8ビットがセットされている文字を、メタ・プレフィックス文字として
# ではなく、直接表示する
set output-meta on

# ある単語について、150を超える補完候補が存在する場合、ユーザに対して
# すべてを表示させたいかどうかを問い合わせる
set completion-query-items 150

# FTP 用
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg

```

```
$endif
```

## 21.4 バインド可能な Readline コマンド

このセクションでは、キー・シーケンスにバインドすることが可能な Readline コマンドについて説明します。

### 21.4.1 移動のためのコマンド

`beginning-of-line (C-a)`

カレント行の先頭に移動します。

`end-of-line (C-e)`

行の末尾に移動します。

`forward-char (C-f)`

1 文字分先に進みます。

`backward-char (C-b)`

1 文字分前に戻ります。

`forward-word (M-f)`

次の単語の末尾へ移動します。単語は、文字と数字により構成されます。

`backward-word (M-b)`

現在カーソルが指している単語、または、1 つ前の単語の先頭に移動します。単語は、文字と数字により構成されます。

`clear-screen (C-l)`

画面を消去し、カレント行を再表示します。その際、カレント行が画面の一番上になるようにします。

`redraw-current-line ()`

カレント行を再表示します。デフォルトでは、このコマンドはバインドされていません。

### 21.4.2 履歴を操作するためのコマンド

`accept-line (Newline, Return)`

カーソルの位置がどこにあっても、その行を受け取ります。この行が空行ではない場合、それを履歴リストに追加します。この行が履歴行である場合は、その履歴行を最初の状態に復元します。

`previous-history (C-p)`

履歴リストを 1 つ上に移動します。

`next-history (C-n)`

履歴リストを 1 つ下に移動します。

`beginning-of-history (M-<)`

履歴の最初の行に移動します。

`end-of-history (M->)`

入力履歴の最後の行、すなわち、現在入力中の行に移動します。

`reverse-search-history (C-r)`

カレント行から始めて上の方向へ検索を行います。必要に応じて履歴の上の方へ移動します。インクリメンタルな検索を行います。

`forward-search-history (C-s)`

カレント行から始めて下の方向へ検索を行います。必要に応じて履歴の下の方へ移動します。インクリメンタルな検索を行います。

`non-incremental-reverse-search-history (M-p)`

カレント行から始めて、必要に応じて履歴の上の方へ移動しつつ、非インクリメンタルな検索を使って、ユーザによって提供された文字列を上の方へ検索します。

`non-incremental-forward-search-history (M-n)`

カレント行から始めて、必要に応じて履歴の下の方へ移動しつつ、非インクリメンタルな検索を使って、ユーザによって提供された文字列を下の方へ検索します。

`history-search-forward ( )`

カレント行の先頭からカレントなカーソル位置 (ポイント) までの間の文字列を、履歴の中で下の方向へ検索します。これは、非インクリメンタルな検索です。デフォルトでは、このコマンドはバインドされていません。

`history-search-backward ( )`

カレント行の先頭からポイントまでの間の文字列を、履歴の中で上の方向へ検索します。これは、非インクリメンタルな検索です。デフォルトでは、このコマンドはバインドされていません。

`yank-nth-arg (M-C-y)`

1 つ前に実行されたコマンドの最初の引数 (通常は、1 つ前の行の 2 つめの単語) を挿入します。引数  $n$  を指定すると、1 つ前に実行されたコマンドの  $n$  番目の単語を挿入します (1 つ前に実行されたコマンドの中の最初の単語を、0 番目の単語とします)。負の値を引数に指定すると、1 つ前に実行されたコマンドの後ろから数えて  $n$  番目の単語を挿入します

`yank-last-arg (M-., M-_)`

1 つ前に実行されたコマンドの最後の引数 (1 つ前の履歴エントリの最後の単語) を挿入します。引数を指定すると、`yank-nth-arg` と同じように動作します。`yank-last-arg` を連続して実行すると、履歴リストを遡って移動していきます。したがって、各行の最後の引数が順番に挿入されていきます。

### 21.4.3 テキストを変更するためのコマンド

`delete-char (C-d)`

カーソル位置にある文字を削除します。カーソルが空行の先頭にあり、最後に入力された文字が `delete-char` にバインドされていない場合は、EOF を返します。

`backward-delete-char` (Rubout)

カーソル位置の前にある文字を削除します。数値引数を指定すると、文字を削除するのではなくキルするよう指示したことになります。

`forward-backward-delete-char` ( )

カーソル位置にある文字を削除します。ただし、カーソルが行末にある場合は、カーソル位置の前にある文字を削除します。デフォルトでは、キーにはバインドされていません。

`quoted-insert` (C-q, C-v)

このコマンドに続けて入力する文字をそのまま行に追加します。これが、例えば `C-q` のようなキー・シーケンスを挿入する方法です。

`tab-insert` (M-TAB)

タブを挿入します。

`self-insert` (a, b, A, 1, !, ...)

その文字自身を挿入します。

`transpose-chars` (C-t)

カーソルの前にある文字をドラッグして、カーソル位置にある文字の後ろに持っていきます。カーソル自身も同様に前進させます。挿入ポイントが行末にある場合には、行の最後の 2 文字を入れ替えます。負の引数を与えても機能しません。

`transpose-words` (M-t)

カーソルの前にある単語をドラッグして、カーソルの後ろにある単語の後ろに持っていきます。カーソル自身も、カーソルの後ろにある単語の後ろに移動します。

`upcase-word` (M-u)

カレントな (あるいは、その 1 つ後ろの) 単語の中のすべての文字を大文字に変換します。負の引数を指定すると、1 つ前の単語の中のすべての文字を大文字に変換しますが、カーソルは移動しません。

`downcase-word` (M-l)

カレントな (あるいは、その 1 つ後ろの) 単語の中のすべての文字を小文字に変換します。負の引数を指定すると、1 つ前の単語の中のすべての文字を小文字に変換しますが、カーソルは移動しません。

`capitalize-word` (M-c)

カレントな (あるいは、その 1 つ後ろの) 単語の先頭文字を大文字に、それ以外の位置にある文字を小文字に変換します。負の引数を指定すると、1 つ前の単語に対して同様の変換を行いますが、カーソルは移動しません。

#### 21.4.4 キルとヤンク

`kill-line` (C-k)

カレントなカーソル位置から行末までのテキストをキルします。

`backward-kill-line` (C-x Rubout)

行の先頭までのテキストをキルします。



`unix-line-discard (C-u)`

カーソル位置から逆方向にカレント行の先頭までをキルします。キルされたテキストは、キル・リングに退避されます。

`kill-whole-line ()`

カーソルの位置にかかわらず、カレント行のすべての文字をキルします。デフォルトでは、バインドされていません。

`kill-word (M-d)`

カーソル位置からカレントな単語の末尾までをキルします。カーソルが単語の間にある場合は、次の単語の末尾までをキルします。単語の境界は、`forward-word`の場合と同様です。

`backward-kill-word (M-DEL)`

カーソルの前にある単語をキルします。単語の境界は、`backward-word`の場合と同様です。

`unix-word-rubout (C-w)`

空白類<sup>3</sup>を単語の境界として、カーソルの前にある単語をキルします。キルされたテキストは、キル・リングに退避されます。

`delete-horizontal-space ()`

ポイントの前後にある、すべての空白（スペース）とタブを削除します。デフォルトでは、バインドされていません。

`kill-region ()`

ポイントとマーク（待避されたカーソル位置）の間のテキストをキルします。このテキストは、領域（*region*）と呼ばれます。デフォルトでは、このコマンドはバインドされていません。

`copy-region-as-kill ()`

領域（*region*）内のテキストを、直ちにヤンクできるよう、キル・バッファにコピーします。デフォルトでは、このコマンドはバインドされていません。

`copy-backward-word ()`

ポイントの前にある単語をキル・バッファにコピーします。単語の境界は、`backward-word`の場合と同様です。デフォルトでは、このコマンドはバインドされていません。

`copy-forward-word ()`

ポイントの後ろにある単語をキル・バッファにコピーします。単語の境界は、`forward-word`の場合と同様です。デフォルトでは、このコマンドはバインドされていません。

`yank (C-y)`

キル・リングの一番上の位置にあるテキストを、バッファ内のカレントなカーソル位置にヤンクします。

`yank-pop (M-y)`

キル・リングを回転させ、新しく一番上の位置にきたテキストをヤンクします。1つ前に実行したコマンドが、`yank` または `yank-pop` であった場合のみ、このコマンドを実行することができます。

---

<sup>3</sup> 訳注：空白（スペース）、水平タブ、改行、垂直タブ、フォーム・フィード

### 21.4.5 数値引数の指定

`digit-argument (M-0, M-1, ... M--)`

既に蓄積済みの引数にこの数字を追加するか、または、この数字によって新しい引数を開始します。負の引数を指定するには、先頭を `(M-)` とします。

`universal-argument ()`

これは、引数を指定する別の方法です。このコマンドの後ろに、場合によって先頭にマイナス記号の付く、1 つ以上の数字が続く場合には、それらの数字が引数を定義します。このコマンドの後ろに数字が続く場合には、`universal-argument` を再実行することによって、その数字引数を終わらせることができます。しかし、このコマンドの後ろに数字が続かない場合の再実行は、無視されます。特殊なケースとして、このコマンドの直後に数字でもマイナス記号でもない文字が続く場合、次に実行されるコマンドの引数カウントは 4 倍されます。引数カウントの初期値は 1 です。したがって、この関数を最初の実行した後は、引数カウントは 4 になり、2 回目に実行した後は 16 になります。以下、同様です。デフォルトでは、キーへのバインドはされていません。

### 21.4.6 Readline による入力補完

`complete (TAB)`

カーソルの前にあるテキストの補完を試みます。これは、アプリケーション固有の動作をします。通常、引数としてファイル名を入力しているときには、ファイル名を補完することができます。コマンド名を入力しているときには、コマンド名を補完することができます。GDB に対してシンボル名を入力しているときには、シンボル名を補完することができます。Bash に対して変数名を入力しているときには、変数名を補完することができます。

`possible-completions (M-?)`

カーソルの前にあるテキストの補完候補を一覧表示します。

`insert-completions (M-*)`

`possible-completions` を実行すれば生成されたであろうテキストの補完候補をすべて、ポイントの前に挿入します。

`menu-complete ()`

`complete` に似ていますが、補完されるべき単語を、補完候補の一覧の中の 1 つと置き換えます。`menu-complete` を繰り返し実行すると、補完候補の一覧から順番に 1 つずつ補完候補が挿入されていきます。候補一覧の終端に達すると、ベル音が鳴らされ、補完前のテキストが復元されます。引数 *n* を指定すると、補完候補の一覧の中で *n* 個先に移動します。一覧を逆方向に戻るために、負の引数を指定することができます。このコマンドは、TAB にバインドすることを意図したのですが、デフォルトではバインドされていません。

`delete-char-or-list ()`

カーソルが行頭、行末のいずれにもない場合、(`delete-char` のように) そのカーソル位置にある文字を削除します。行末にある場合は、`possible-completions` と同一の振る舞いします。このコマンドは、デフォルトではバインドされていません。

### 21.4.7 キーボード・マクロ

`start-kbd-macro (C-x (`

カレントなキーボード・マクロの構成要素として入力される文字の保存を開始します。

`end-kbd-macro (C-x ))`

カレントなキーボード・マクロの構成要素として入力された文字の保存を終了して、そのキーボード・マクロの定義を保存します。

`call-last-kbd-macro (C-x e)`

最後に定義されたキーボード・マクロを再実行します。マクロの中の文字群が、あたかもキーボードから入力されたかのように、現われます。

### 21.4.8 その他のコマンド

`re-read-init-file (C-x C-r)`

`inputrc` ファイルの内容を読み込み、その中にあるバインディングや変数割り当てをすべて組み込みます。

`abort (C-g)`

カレントな編集コマンドの実行を停止し、( `bell-style` の設定次第では ) 端末のベル音を鳴らします。

`do-uppercase-version (M-a, M-b, M-x, ...)`

メタ化された文字 `x` が小文字である場合、対応する大文字にバインドされているコマンドを実行します。

`prefix-meta (ESC)`

次に入力される文字をメタ化します。これは、メタ・キーのないキーボード用のコマンドです。‘`ESC f`’を入力するのは、‘`M-f`’を入力するのと同じことです。

`undo (C-_, C-x C-u)`

インクリメンタルな取り消し処理を実行します。取り消す内容は、各行ごとに別々に記憶されています。

`revert-line (M-r)`

行に加えられたすべての変更を取り消します。これは、`undo` コマンドを、行を元の状態に戻すのに必要な回数繰り返して実行するようなものです。

`tilde-expand (M-~)`

カレントな単語に対して、チルダ展開を実行します。

`set-mark (C-@)`

カレントなポイントにマークをセットします。数値引数があれば、その位置にマークがセットされます。

`exchange-point-and-mark (C-x C-x)`

ポイントとマークを交換します。待避されていた位置がカレントなカーソル位置としてセットされ、元のカーソル位置はマークとして待避されます。

**character-search (C-])**

文字を 1 つ読み取り、その文字が次に現われるところにポイントを移動します。負の数を指定すると、その文字が以前に現われたところを探します。

**character-search-backward (M-C-])**

文字を 1 つ読み取り、その文字が前に現われたところにポイントを移動します。負の数を指定すると、その文字が次に現われるところを探します。

**insert-comment (M-#)**

カレント行の先頭に `comment-begin` 変数の値が挿入され、挿入後の行が、あたかも改行が入力されたかのように、受け付けられます。

**dump-functions ()**

Readline の出力ストリームに、すべての関数とそのキー・バインディングを出力します。数値引数が指定されると、`inputrc` ファイルの一部として使用することのできる形式に、出力がフォーマットされます。このコマンドは、デフォルトではバインドされていません。

**dump-variables ()**

Readline の出力ストリームに、値をセットすることのできるすべての変数とその値を出力します。数値引数が指定されると、`inputrc` ファイルの一部として使用することのできる形式に、出力がフォーマットされます。このコマンドは、デフォルトではバインドされていません。

**dump-macros ()**

マクロにバインドされているすべての Readline キー・シーケンスと、そのキー・シーケンスが出力する文字列を出力します。数値引数が指定されると、`inputrc` ファイルの一部として使用することのできる形式に、出力がフォーマットされます。このコマンドは、デフォルトではバインドされていません。

## 21.5 Readline の vi モード

Readline ライブラリは、vi の編集機能のフルセットを提供してはいませんが、簡単な行編集を行うのに十分な機能は備えています。Readline の vi モードは、POSIX 1003.2 標準にしたがって動作します。

emacs 編集モードと vi 編集モードを対話的に切り替えるには、コマンド `M-C-j` (`toggle-editing-mode`) を使用してください。Readline のデフォルトは emacs モードです。

vi モードで行入力を行うときには、あたかも 'i' を入力したかのように、最初から「挿入」モードになっています。`(ESC)` を押すと「コマンド」モードになり、標準的な vi の移動キーによって行のテキストを編集することができます。すなわち、'k' により前の履歴行に移動すること、'j' によって後ろの履歴行に移動すること、などが可能です。

## 22 対話的な履歴の使い方

ここではユーザの見地に立って、GNU 履歴ライブラリの対話的な使い方を説明します。これはユーザズ・ガイドのようなものであると考えてください。

### 22.1 履歴展開

履歴ライブラリは、`cs`hの履歴展開機能に似た機能を提供します。このセクションでは、履歴情報を操作するための構文を説明します。

履歴展開によって、履歴リストの中にある単語列を入力ストリームの中に組み入れることができます。これによって、コマンドの再実行、以前に実行したコマンドにおける引数のカレントな入力行への挿入、また、以前に実行したコマンドにおける誤りの機敏な修正が簡単にできるようになります。

履歴展開は 2 つの処理から構成されます。第 1 の処理は、履歴リストのどの行を置換の際に使用するべきかを決定することです。第 2 の処理は、この行のうちどの部分がカレント行に組み込まれるかを決定することです。履歴から選択される行をイベントと呼び、処理対象となる部分のことをワードと呼びます。選択されたワード群を操作するために、いくつかの修飾子が利用できるようになっています。行は、`bash`と同様の方法によってワードに分割されます。したがって、引用符によって囲まれた複数の単語 (ワード) は 1 つのワードとみなされます。履歴展開は、履歴展開文字の出現によって開始されます。履歴展開文字は、デフォルトでは `!` です。

#### 22.1.1 イベント指定子

イベント指定子とは、履歴リスト内のコマンド行エントリへの参照です。

- `!`           履歴置換を開始します。ただし次に続く文字が、空白、タブ、行末、`=`、`(`である場合は例外です。
- `!n`        コマンド行番号 *n* のコマンド行を参照します。
- `!-n`       *n* 行前のコマンド行を参照します。
- `!!`        1 つ前のコマンドを参照します。これは `!-1` と同義です。
- `!string`    コマンドの先頭が文字列 *string* で始まるコマンドのうち、最後に実行されたものを参照します。
- `!?string[?]`   文字列 *string* を含むコマンドのうち、最後に実行されたものを参照します。*string* の直後が改行である場合は、末尾の `?` を省略することができます。
- `^string1^string2^`   クイック置換を実行します。最後に実行されたコマンドの *string1* の部分を *string2* に置き換えて実行します。これは、`!!:s/string1/string2/` と同じことです。
- `!#`        それまでに入力されたコマンド行全体を指します。

#### 22.1.2 ワード指定子

ワード指定子は、イベントから希望するワードを選択する目的で使われます。コロン (`:`) が、イベント指定子とワード指定子の区切り文字になります。ワード指定子が `^`、`$`、`*`、`-`、`%`

で始まる場合は、この区切り文字は省略することができます。ワードは行の先頭から番号が付与され、最初のワードは 0 (ゼロ) 番になります。複数のワードがカレント行に組み込まれる際には、ワードの間は単一の空白で区切られます。

- 0 (zero)    ゼロ番目のワードです。多くのアプリケーションにおいて、これはコマンド・ワードです。
- n*            *n* 番目のワードです。
- ^            最初の引数、すなわち 1 番目のワードです。
- \$            最後の引数です。
- %            最後に実行された '*?string?*' 検索にマッチしたワードです。
- x-y*          ある範囲のワードを指します。'*-y*' は、'*0-y*' の省略形です。
- \*            ゼロ番目のワードを除く、すべてのワードです。これは '*1-\$*' と同義です。イベントの内部にワードが 1 つしか存在しない場合に '\*' を使っても、エラーにはなりません。この場合には、空の文字列が返されます。
- x\**           '*x-\$*' の省略形です。
- x-*           '*x\**' のように '*x-\$*' を省略したのですが、最後のワードは除外されます。

ワード指定子にイベント指定子の指定がない場合、1 つ前のコマンドがイベントとして使われます。

### 22.1.3 修飾子

ワード指定子は必須ではありませんが、それを指定した場合、その後ろに以下の修飾子を 1 つ以上連結して付与することができます。個々の修飾子の前にはコロン (':') を付けます。

- h*            パス名の末尾の部分を削除したヘッド部です。
- t*            パス名の末尾の部分を残し、それより前にある部分をすべて削除します。
- r*            '*.suffix*' という形式の拡張子を削除したベース名です。
- e*            拡張子以外のすべての部分を削除します。
- p*            新しいコマンドを表示するだけで実行しません。
- s/old/new/*    イベント行において最初に出現した *old* を *new* によって置換します。 '/' の代わりに任意の区切り文字を使うことができます。単一のバックスラッシュを使うことによって、*old* と *new* の中においてこの区切り文字を通常の文字として引用することができます。*new* の中に '&' があると、それは *old* によって置き換えられます。単一のバックスラッシュを使うことによって、'&' を通常の文字として引用することができます。末尾の区切り文字は、それが入力行の最後の文字であれば、省略可能です。
- &*            1 つ前に実行された置換を繰り返します。
- g*            変更が、イベント行全体において適用されるようにします。*gs/old/new/* のように、's' と組み合わせて使います。また、'&' と組み合わせて使うこともできます。

## 付録 A ドキュメントのフォーマット

GDB 5 には、PostScript または GhostScript でそのまま印刷できる、フォーマット済みのリファレンス・カードが含まれています。<sup>1</sup> これは、メインのソース・ディレクトリの下に 'gdb' サブディレクトリにあります。PostScript または Ghostscript を使えるプリンタがあれば、'refcard.ps' を使ってすぐにリファレンス・カードを印刷することができます。

GDB 5 には、リファレンス・カードのソースも含まれています。T<sub>E</sub>X を使えば、以下のようにしてこれをフォーマットすることができます。

```
make refcard.dvi
```

GDB のリファレンス・カードは、米国のレター・サイズ用の紙にランドスケープ・モードで印刷するようにデザインされています。レター・サイズは、横幅が 11 インチ、高さが 8.5 インチです。DVI 出力プログラムへのオプションとして、この印刷形式を指定する必要があります。

すべての GDB ドキュメントは、マシン上で読むことのできるディストリビューションの一部として提供されます。ドキュメントは Texinfo フォーマットで記述されています。これは、単一のソースからオンライン・マニュアルとハードコピー・マニュアルの両方を生成するドキュメント・システムです。Info フォーマット・コマンドの 1 つを使ってオンライン・ドキュメントを作成することができます。T<sub>E</sub>X (または texi2roff) を使ってハード・コピーの組版ができます。

GDB には、このマニュアルのフォーマット済みのオンライン Info バージョンも含まれています。これは、'gdb' サブディレクトリにあります。メインの Info ファイルは 'gdb-5.0/gdb/gdb.info' で、同じディレクトリにある 'gdb.info\*' というパターンにマッチする従属ファイルを参照します。必要であれば、これらのファイルを印刷したり、任意のエディタで表示して読むこともできます。しかし、これらのファイルは、GNU Emacs の info サブシステムや GNU Texinfo の一部として配布されるスタンドアロンの info プログラムを使った方が読みやすいでしょう。

これらの Info ファイルを自分でフォーマットしたいのであれば、texinfo-format-buffer や makeinfo のような Info フォーマット・プログラムが必要になります。

makeinfo がインストールされていて、GDB ソース・ディレクトリのトップ・レベル (バージョン 5.0 では 'gdb-5.0') にいる場合は、以下のようにして Info ファイルを作成することができます。

```
cd gdb
make gdb.info
```

このマニュアルのコピーの組版を行って印刷するには、T<sub>E</sub>X、T<sub>E</sub>X の DVI 出力ファイルを印刷するプログラム、および、Texinfo 定義ファイル 'texinfo.tex' が必要です。

T<sub>E</sub>X は組版プログラムです。T<sub>E</sub>X は直接ファイルを印刷しませんが、DVI ファイルと呼ばれるものを生成します。組版されたドキュメントを印刷するには、DVI ファイルを印刷するプログラムが必要です。システム上に T<sub>E</sub>X がインストールされていれば、DVI ファイルを印刷するプログラムも入っている可能性があります。印刷に使われるコマンドの正確な名前はシステムにより異なります。lpr -d が一般によく使われます。また (PostScript プリンタでは) dvips がよく使われます。DVI プリント・コマンドを使う際には、ファイル名に拡張子を付けないか、あるいは、'.dvi' という拡張子を付ける必要があるかもしれません。

また、T<sub>E</sub>X は 'texinfo.tex' という名のマクロ定義ファイルを必要とします。このファイルは T<sub>E</sub>X に対して、Texinfo フォーマットで記述されたドキュメントをどのようにして組版するかを教えます。T<sub>E</sub>X は自分自身では、Texinfo ファイルを読むことも組版することもできません。

<sup>1</sup> 原注: バージョン 5.0 では 'gdb-5.0/gdb/refcard.ps' です。

‘texinfo.tex’は GDB とともに配布されていて、‘gdb-version-number/texinfo’ディレクトリにあります。

T<sub>E</sub>X と DVI 印刷プログラムがインストールされていれば、このマニュアルを組版して、印刷することができます。メインのソース・ディレクトリの下の ‘gdb’サブディレクトリ（例えば、‘gdb-5.0/gdb’）に移動して、以下のように実行します。

```
make gdb.dvi
```

その後、‘gdb.dvi’を DVI 印刷プログラムに渡します。



## 付録 B GDB のインストール

GDB には、インストールのための準備作業を自動化する `configure` スクリプトが付属しています。`configure` を実行した後に `make` を実行することで、`gdb` をビルドすることができます。<sup>1</sup> GDB ディストリビューションには、GDB をビルドするのに必要なすべてのソース・コードが、単一のディレクトリの下に収められています。このディレクトリの名前は通常、`'gdb'` の後ろにバージョン番号を付加したものです。

例えば、バージョン 5.0 の GDB ディストリビューションは、`'gdb-5.0'` というディレクトリに収められています。このディレクトリには以下のものが含まれます。

```
gdb-5.0/configure ( およびサポート・ファイル )
    GDB、および、GDB が必要とするすべてのライブラリの構成を行うためのスクリプト

gdb-5.0/gdb
    GDB 自身に固有のソース

gdb-5.0/bfd
    Binary File Descriptor ライブラリのソース

gdb-5.0/include
    GNU インクルード・ファイル

gdb-5.0/libiberty
    '-liberty' フリー・ソフトウェア・ライブラリのソース

gdb-5.0/opcodes
    opcode テーブルおよび逆アセンブラのライブラリのソース

gdb-5.0/readline
    GNU コマンドライン・インターフェイスのソース

gdb-5.0/glob
    GNU ファイル名パターン・マッチング・サブルーチンのソース

gdb-5.0/malloc
    メモリにマップされる GNU malloc パッケージのソース
```

GDB の構成とビルドを行う最も簡単な方法は、`'gdb-version-number'` ソース・ディレクトリから `configure` を実行することです。ここでの例では、このディレクトリは `'gdb-5.0'` です。

もしまだ `'gdb-version-number'` ソース・ディレクトリにいないのであれば、まずそこに移動してください。続いて `configure` を実行します。GDB が実行されるプラットフォームの識別子を引数として渡します。

例えば、以下のようにします。

```
cd gdb-5.0
./configure host
make
```

---

<sup>1</sup> 原注：バージョン 5.0 よりさらに新しい GDB を持っている場合には、ソースの中に含まれる `'README'` ファイルを参照してください。このマニュアルの出版後、インストール手順が改善されていることがあるかもしれません。

*host* は、GDB が実行されるプラットフォームを識別する識別子です。例えば *'sun4'* や *'decstation'* などです (多くの場合 *host* は省略することができます。この場合 *configure* は、ユーザのシステムを調べることによって正しい値を推定しようとします)。

*'configure host'* を実行した後に *make* を実行することで、*'bfd'*、*'readline'*、*'mmapalloc'*、*'libiberty'* の各ライブラリがビルドされ、最後に *gdb* 自体がビルドされます。構成されたソース・ファイルやバイナリは、対応するソース・ディレクトリに残されます。*configure* は Bourne シェル ( */bin/sh* ) のスクリプトです。ユーザが別のシェルを実行していて、システムがこのことを自動的に認識してくれない場合は、明示的に *sh* にスクリプトを実行させる必要があるかもしれません。

```
sh configure host
```

バージョン 5.0 のソース・ディレクトリである *'gdb-5.0'* のように、配下に複数のライブラリやプログラムのソース・ディレクトリを含むディレクトリから *configure* を実行すると、*configure* は配下にあるそれぞれのディレクトリのための構成ファイルを作成します ( *'--norecursion'* オプションによって、そうしないよう指定した場合は別です )。

GDB ディストリビューションの中のある特定のサブディレクトリを構成したいだけの場合には、そのサブディレクトリから *configure* スクリプトを実行することができます。ただし、*configure* スクリプトへのパスを必ず指定してください。

例えば、バージョン 5.0 では、*bfd* サブディレクトリだけを構成するには以下のようにします。

```
cd gdb-5.0/bfd
../configure host
```

*gdb* はどこにでもインストールできます。あらかじめ固定されたパスは 1 つもありません。ただし、ユーザのパスにある ( *'SHELL'* 環境変数により指定される ) シェルが誰にでも読み取り可能であることを確かめる必要があります。GDB はシェルを使ってユーザ・プログラムを起動するというのを憶えておいてください。子プロセスが読み取り不可のプログラムである場合、システムによっては、GDB がそれをデバッグするのを拒否します。

## B.1 別ディレクトリでの GDB のコンパイル

いくつかのホスト・マシンおよびターゲット・マシン用の GDB を実行したい場合、ホストとターゲットの個々の組み合わせ用にコンパイルされた異なる *gdb* が必要になります。*configure* には、個々の構成をソース・ディレクトリではなく個別のサブディレクトリに生成する機能があり、このようなことが簡単にできるように設計されています。ユーザの使っている *make* プログラムに *'VPATH'* 機能があれば ( *GNU make* にはあります )、これら個々のディレクトリにおいて *make* を実行することで、そこで指定されている *gdb* プログラムをビルドすることができます。

個別のディレクトリにおいて *gdb* をビルドするには、ソースの置かれている場所を指定するために、*'--srcdir'* オプションを使って *configure* を実行します (同時に、ユーザの作業ディレクトリから *configure* を見つけるためのパスも指定する必要があります。もし、*configure* へのパスが *'--srcdir'* への引数として指定するものと同じであれば、*'--srcdir'* オプションは指定しなくてもかまいません。指定されなければ、同じであると仮定されます)。

例えば、バージョン 5.0 で Sun 4 用の GDB を別のディレクトリにおいてビルドするには、以下のようにします。

```
cd gdb-5.0
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-5.0/configure sun4
make
```

configureが、別の場所にあるソース・ディレクトリを使って、ある構成を作成する際には、ソース・ディレクトリ配下のディレクトリ・ツリーと同じ構造のディレクトリ・ツリーを（同じ名前で）バイナリ用に作成します。この例では、Sun 4 用のライブラリ ‘libiberty.a’は ‘gdb-sun4/libiberty’ディレクトリに、GDB 自身は ‘gdb-sun4/gdb’ディレクトリにそれぞれ作成されます。

いくつかの GDB の構成を個別のディレクトリにおいてビルドする理由としてよくあるのが、クロス・コンパイル（GDB はホストと呼ばれるあるマシン上で動作し、ターゲットと呼ばれる別のマシンで実行されているプログラムをデバッグする）環境用に GDB を構成する場合です。クロス・デバッグのターゲットは、configureに対する ‘--target=target’ オプションを使って指定します。

プログラムやライブラリをビルドするために makeを実行するときには、構成されたディレクトリにいなければなりません。これは、configureを実行したときにいたディレクトリ（または、そのサブディレクトリの1つ）です。

configureが個別のソース・ディレクトリに生成した Makefileは再帰的に呼び出されます。‘gdb-5.0’（あるいは、‘--srcdir=dirname/gdb-5.0’により構成された別のディレクトリ）などのソース・ディレクトリにおいて makeを実行すると、必要とされるすべてのライブラリがビルドされ、その後に GDB がビルドされることになります。

複数のホストまたはターゲットの構成が、異なる複数のディレクトリに存在する場合、（例えば、それらが個々のホスト上に NFS マウントされている場合）並行して makeを実行することができます。複数の構成が互いに干渉し合うということはありません。

## B.2 ホストとターゲットの名前の指定

configureスクリプトにおけるホストおよびターゲットの指定方法は、3つの名称部分を持ちますが、あらかじめ定義された短い別名もいくつかサポートされています。完全名は、以下のようなパターンの3つの情報部分を持ちます。

*architecture-vendor-os*

例えば、ホストを指定する引数 *host* として、あるいは、--target=target オプションの *target* の部分に、sun4という別名を使うことができます。これと同等の完全名は ‘sparc-sun-sunos4’ です。

GDB に付属している configureスクリプトには、サポートされているすべてのホスト名、ターゲット名、および、別名を問い合わせするための機能はありません。configureは、Bourne シェル・スクリプトの config.subを呼び出すことによって、省略名を完全名に対応付けします。もしそうしたいのであれば、このスクリプトを読むことによって、あるいは、このスクリプトを使うことによって、省略名の意味が推測と合っているかどうかをテストすることもできます。以下に例を示します。

```
% 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も、GDB ディストリビューションの一部としてソース・ディレクトリ (バージョン 5.0 では、'gdb-5.0') に入っています。

### B.3 configure オプション

以下に、GDB をビルドする上でほとんどの場合に役に立つ configure のオプションと引数の要約を示します。configure には、ここには挙げられていないオプションもいくつかあります。configure に関する完全な説明については、See Info ファイル 'configure.info', node 'What Configure Does'。

```
configure [--help]
          [--prefix=dir]
          [--exec-prefix=dir]
          [--srcdir=dirname]
          [--norecursion] [--rm]
          [--target=target]
          host
```

そうしたいのであれば、'---'ではなく単一の '-' でオプションを始めることもできますが、'---'を使うとオプション名を省略することができます。

--help      configure の実行方法の簡単な要約を表示します。

--prefix=dir  
プログラムおよびファイルをディレクトリ 'dir' にインストールするようソースを構成します。

--exec-prefix=dir  
プログラムをディレクトリ 'dir' にインストールするようソースを構成します。

--srcdir=dirname  
注意：このオプションを使うには、GNU make、あるいは、VPATH 機能を持つ他の make を使用する必要があります。  
GDB ソース・ディレクトリとは別のディレクトリに構成を作成する場合に、このオプションを使用します。特に、いくつかの構成を個別のディレクトリにおいて同時に作成 (かつ維持) する場合に、このオプションを使うことができます。configure は、構成に固有のファイルをカレント・ディレクトリに書き込みますが、dirname ディレクトリにあるソースを使うように、それらのファイルの内容を調整します。configure は、dirname ディレクトリ配下のソース・ディレクトリ・ツリーと同じ構造を持つディレクトリ・ツリーを、作業ディレクトリの下に作成します。

--norecursion  
configure が実行されたディレクトリ・レベルだけを構成します。サブディレクトリまで含めて構成することはしません。

--target=target  
指定されたターゲット target で実行するプログラムをクロス・デバッグするために、GDB を構成します。このオプションを指定しないと、GDB と同じマシン (ホスト) で実行されるプログラムをデバッグするよう、GDB は構成されます。  
利用可能なすべてのターゲットの一覧を生成する、便利な方法はありません。

*host* ... 指定されたホスト *host* 上で実行されるよう GDB を構成します。  
利用可能なすべてのホストの一覧を生成する、便利な方法はありません。

ほかにも利用可能な多くのオプションがありますが、これは通常、特殊な目的にのみ必要とされるものです。



## インデックス

- リモート・シリアル・プロトコル ..... 123
- リファレンス・カード ..... 261
- プロセスのイメージ ..... 139
- ドキュメント ..... 261
- デバッグのクラッシュ ..... 237
- コマンド・ファイル ..... 165
- クラッシュ、デバッグの ..... 237
- インストール ..... 263
- 呼び出し、オーバーロードされた関数の  
[よびだし、オーバーロードされたかん  
すうの] ..... 87
- 選択、ターゲットのバイト・オーダの [せん  
たく、ターゲットのバイト・オーダの]  
..... 117
- 消去、ブレイクポイント、ウォッチポイン  
ト、キャッチポイントの [しょうきょ、  
ブレイクポイント、ウォッチポイント、  
キャッチポイントの] ..... 39
- 削除、ブレイクポイント、ウォッチポイン  
ト、キャッチポイントの [さくじょ、ブ  
レイクポイント、ウォッチポイント、  
キャッチポイントの] ..... 39
- 例外のキャッチ、アクティブ・ハンドラの一  
覧表示 [れいがいのキャッチ、アクティ  
ブ・ハンドラのいちらんひょうじ] .. 55
- 概要、リモート・シリアル・デバッグ処理  
の [がいよう、リモート・シリアル・デ  
バッグしよりの] ..... 118
- フォーマットのオプション ..... 70
- ディレクトリ、ソース・ファイルの .... 59
- ディレクトリ、コンパイルする ..... 59
- オーバーロードされている関数のブレイク  
[オーバーロードされているかんすうの  
ブレイク] ..... 88
- リターン ..... 107
- メモリにマップされたシンボル・ファイル  
..... 110
- パッチ、バイナリの ..... 108
- バイナリのパッチ ..... 108
- シンボル・ファイル、メモリにマップされ  
た ..... 110
- バイト・オーダの選択、ターゲットの [バイ  
ト・オーダのせんたく、ターゲットの]  
..... 117
- オーバーロードされた関数の呼び出し  
[オーバーロードされたかんすうのよび  
だし] ..... 87
- リモート・シリアル・スタブの一覧 [リモ  
ート・シリアル・スタブのいちらん] .. 119
- デバッグ・ターゲット ..... 115
- ダイナミック・リンク ..... 111
- ターゲットのバイト・オーダの選択 [ター  
ゲットのバイト・オーダのせんたく]  
..... 117
- シリアル回線速度、日立マイクロ [シリア  
ルかいせんそくど、ひたちマイクロ]  
..... 145
- エラー、正当な入力にたいする [エラー、せ  
いとうなにゅうりよくにたいする]  
..... 237
- ブレイクポイント・コマンド、GDB/MI [ブレ  
イクポイント・コマンド、GDB/MI]  
..... 185
- シリアル装置、日立マイクロ [シリアルそ  
うち、ひたちマイクロ] ..... 145
- シリアル接続、デバッグ用 [シリアルせつぞ  
く、デバッグよう] ..... 161
- カーネル・オブジェクト ..... 136

|                                                             |     |                                                      |     |
|-------------------------------------------------------------|-----|------------------------------------------------------|-----|
| カーネル・オブジェクト表示.....                                          | 136 | アクティブ・ターゲット.....                                     | 115 |
| 要約、リモート・シリアル・デバッグ処理<br>の [ようやく、リモート・シリアル・デ<br>バッグしよりの]..... | 121 | 例外のハンドラ、一覧表示方法 [れいがいの<br>ハンドラ、いちらんひょうじほうほう]<br>..... | 55  |
| リモート・スタブ、サポート・ルーチン<br>.....                                 | 120 | 保存、シンボル・テーブルの [ほぞん、シン<br>ボル・テーブルの].....              | 110 |
| リモート・シリアル・スタブ、メイン・<br>ルーチン.....                             | 119 | 註釈、プログラムの実行 [ちゅうしゃく、プ<br>ログラムのじっこう].....             | 177 |
| リモート・デバッグ処理、スタブの実例 [リ<br>モート・デバッグしより、スタブのじ<br>つれい].....     | 123 | リダイレクト.....                                          | 24  |
| 初期化、リモート・シリアル・スタブの<br>[しょきか、リモート・シリアル・スタ<br>ブの].....        | 119 | メモリのトレース.....                                        | 31  |
| 割り込み、リモート・プログラムの [わりこ<br>み、リモート・プログラムの].....                | 122 | マルチプロセス.....                                         | 28  |
| リモート・プログラムの割り込み [リモー<br>ト・プログラムのわりこみ].....                  | 122 | マルチスレッド.....                                         | 26  |
| リモート・ターゲットの割り込み [リモー<br>ト・ターゲットのわりこみ].....                  | 120 | プログラムの実行、註釈 [プログラムのじっ<br>こう、ちゅうしゃく].....             | 177 |
| リモート・シリアル・スタブ.....                                          | 119 | 註釈、ブレイクポイント [ちゅうしゃく、ブ<br>レイクポイント].....               | 176 |
| スタブを使わないリモート接続 [スタブを<br>つかわないリモートせつぞく].....                 | 134 | ブレイクポイント.....                                        | 31  |
| 実例、リモート・スタブの [じつれい、リ<br>モート・スタブの].....                      | 123 | ブレイクポイントとスレッド.....                                   | 49  |
| 実例、スタブのデバッグの [じつれい、スタ<br>ブのデバッグの].....                      | 123 | ブレイクポイント・コマンド.....                                   | 43  |
|                                                             |     | ブレイクポイント、註釈 [ブレイクポイン<br>ト、ちゅうしゃく].....               | 176 |
|                                                             |     | フレーム・ポインタ.....                                       | 51  |
|                                                             |     | バックトレース.....                                         | 52  |
|                                                             |     | スレッド番号.....                                          | 27  |
|                                                             |     | スレッドのブレイクポイント.....                                   | 49  |
|                                                             |     | シンボルのオーバーロード.....                                    | 44  |
|                                                             |     | ブレイクポイント、ウォッチポイント、<br>キャッチポイントの削除 [ブレイクポイ            |     |



- ント、ウォッチポイント、キャッチポイントのさくじょ] ..... 39
- ハードウェア・ウォッチポイント ..... 36
- ソフトウェア・ウォッチポイント ..... 36
  
- 作業ディレクトリ、ユーザ・プログラムの  
[さぎょうディレクトリ、ユーザ・プログラムへの] ..... 24
  
- ユーザ・プログラムの作業ディレクトリ  
[ユーザ・プログラムのさぎょうディレクトリ] ..... 24
- ブレイクポイント、条件付きの [ブレイクポイント、じょうけんつきの] ..... 41
- スレッドとウォッチポイント ..... 37
  
- 範囲指定、ブレイクポイントの [はんいし  
てい、ブレイクポイントの] ..... 32
  
- 引数、ユーザ・プログラムへの [ひきすう、  
ユーザ・プログラムへの] ..... 22
  
- スレッド識別子 (システム) HP-UX[ス  
レッドしきべつし (システム)  
HP-UX] ..... 27
  
- 内部のブレイクポイント番号 [ないぶのブ  
レイクポイントばんごう] ..... 35
  
- 条件付きのブレイクポイント [じょうけん  
つきのブレイクポイント] ..... 41
  
- 環境、ユーザ・プログラムの [かんきょう、  
ユーザ・プログラムの] ..... 23
  
- 識別子、システムのスレッド [しきべつし、  
システムのスレッド] ..... 26
  
- ユーザ・プログラムへの引数 [ユーザ・プロ  
グラムへのひきすう] ..... 22
- スレッド識別子、システムの [スレッドしき  
べつし、システムの] ..... 26
- コア・ファイルへの書き込み [コア・ファ  
イルへのかきこみ] ..... 108
- ユーザ・プログラムの環境 [ユーザ・プロ  
ラムのかんきょう] ..... 23
  
- 書き込み、実行コードへの [かきこみ、じっ  
こうコードへの] ..... 108
  
- パイプ ..... 22
- デバッグ、最適化コードの [デバッグ、さい  
てきかコードの] ..... 21
- システムのスレッド識別子 [システムのス  
レッドしきべつし] ..... 26
- シリアル・プロトコル、GDB リモート  
..... 123
- ストリーム・レコード、GDB/MI[ストリー  
ム・レコード、GDB/MI] ..... 183
  
- 註釈、失効メッセージ [ちゅうしゃく、しっ  
こうメッセージ] ..... 177
  
- 失効メッセージ、註釈 [しっこうメッセー  
ジ、ちゅうしゃく] ..... 177

|                                                   |     |                                                     |     |
|---------------------------------------------------|-----|-----------------------------------------------------|-----|
| レジスタ・スタック、AMD29K の .....                          | 156 | スコープ .....                                          | 93  |
| シェル・エスケープ .....                                   | 13  | コマンド・フック .....                                      | 164 |
| オーバーロードされた関数 [オーバーロー<br>ドされたかんすう] .....           | 89  | シンボルの即時読み込み [シンボルのそく<br>じよみこみ] .....                | 110 |
| 読み込み、シンボルの即時 [よみこみ、シン<br>ボルのそくじ] .....            | 110 | シンボル・テーブル .....                                     | 109 |
|                                                   |     | ダウンロード、日立 SH への [ダウンロー<br>ド、ひたち SH への] .....        | 145 |
|                                                   |     | シンボル・ダンプ .....                                      | 103 |
|                                                   |     | コア・ダンプ・ファイル .....                                   | 109 |
| 部分的シンボル・ダンプ [ぶぶんてきシンボ<br>ル・ダンプ] .....             | 103 | 書き込み、コア・ファイルへの [かきこみ、<br>コア・ファイルへの] .....           | 108 |
| 実行コードへの書き込み [じっこうコード<br>へのかきこみ] .....             | 108 | コロンの、スコープ演算子の 2 重 [コロンの、ス<br>コープえんざんしの 2 じゅう] ..... | 93  |
| 註釈、ソースの表示 [ちゅうしゃく、ソース<br>のひょうじ] .....             | 178 | 浮動小数ハードウェア [ふどうしょうすう<br>ハードウェア] .....               | 78  |
| リロード .....                                        | 102 | 再ロード、シンボルの [さいロード、シンボ<br>ルの] .....                  | 102 |
| ソースの表示、註釈 [ソースのひょうじ、<br>ちゅうしゃく] .....             | 178 | オブジェクトの形式と C++ [オブジェクトの<br>けいしきと C++] .....         | 86  |
| ユーザ定義コマンド [ユーザていぎコマン<br>ド] .....                  | 163 |                                                     |     |
| フック、コマンド用 [フック、コマンドよう]<br>.....                   | 164 | 浮動小数点レジスタ [ふどうしょうすうて<br>んレジスタ] .....                | 77  |
| 出力構文の変更案、GDB/MI [しゅつりょくこ<br>うぶんのへんこうあん、GDB/MI] .. | 233 | 併用、ターゲットの [へいよう、ターゲッ<br>トの] .....                   | 115 |
| 変数オブジェクト、GDB/MI [へんすうオブ<br>ジェクト、GDB/MI] .....     | 230 |                                                     |     |

- 一時停止、出力の [いちじていし、しゅつ  
りょくの] ..... 159
- レジスタ ..... 77
- デマングル ..... 73
- ソースのパス ..... 59
- シンボリック形式のアドレス解釈 [シンボ  
リックけいしきのアドレスかいしゃく]  
..... 71
- コンパイルするディレクトリ ..... 59
- 参照する位置、ポインタの [さんしょうする  
いち、ポインタの] ..... 71
- ポインタの参照する位置 [ポインタのさん  
しょうするいち] ..... 71
- キャストしたメモリ ..... 63
- カレントなディレクトリ ..... 59
- 命令の表示、アセンブリ [めいれいのひょう  
じ、アセンブリ] ..... 60
- アセンブリ命令の表示 [アセンブリめいれ  
いのひょうじ] ..... 60
- アセンブリ言語の選択 [アセンブリげんご  
のせんたく] ..... 61
- ディレクトリ、現在の [ディレクトリ、げん  
ざいの] ..... 59
- エンコードされた形式 [エンコードされた  
けいしき] ..... 73
- 表示、マシン命令の [ひょうじ、マシンめい  
れいの] ..... 60
- 選択、マシン命令の [せんたく、マシンめい  
れいの] ..... 61
- フォーマット、出力 [フォーマット、しゅつ  
りょく] ..... 66
- 変数値、正しくない [へんすうち、ただしく  
ない] ..... 64
- メモリ、型変換した [メモリ、かたへんかん  
した] ..... 63
- 現在のディレクトリ [げんざいのディレク  
トリ] ..... 59
- コンビニエンス変数 [コンビニエンスへん  
すう] ..... 76
- オーバーロード 解決 [オーバーロードかい  
けつ] ..... 89
- スレッド、停止した [スレッド、ていしし  
た] ..... 49
- 機械語命令の表示 [きかいごめいれいのひ  
ょうじ] ..... 60
- 機械語命令の選択 [きかいごめいれいのせ  
んたく] ..... 61
- コマンド行の編集 [コマンドぎょうのへん  
しゅう] ..... 241

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>           註釈、プロンプト [ちゅうしゃく、プロンプト] ..... 175         </p> <p>           出力フォーマット [しゅつりよくフォーマット] ..... 66         </p> <p>           マシン命令の表示 [マシンめいれいのひょうじ] ..... 60<br/>           マシン命令の選択 [マシンめいれいのせんたく] ..... 61<br/>           プロンプト、註釈 [プロンプト、ちゅうしゃく] ..... 175         </p> <p>           復帰、関数からの [ふっき、かんすうからの] ..... 107         </p> <p>           型変換したメモリ [かたへんかんしたメモリ] ..... 63         </p> <p>           初期化ファイル名 [しょきかファイルめい] ..... 165         </p> <p>           作業ディレクトリ [さぎょうディレクトリ] ..... 59         </p> <p>           呼び出し、関数の [よびだし、かんすうの] ..... 108         </p> <p>           名前、シンボルの [なまえ、シンボルの] ..... 101         </p> | <p>           確認メッセージ、GDB リモートの .... 123         </p> <p>           ターゲット出力、GDB/MI [ターゲットしゅつりよく、GDB/MI] ..... 182<br/>           ステータス出力、GDB/MI [ステータスしゅつりよく、GDB/MI] ..... 181<br/>           コンソール出力、GDB/MI [コンソールしゅつりよく、GDB/MI] ..... 181         </p> <p>           帯域外レコード、GDB/MI [たいいきがイレコード、GDB/MI] ..... 184         </p> <p>           メモリ・モデル、H8/500 ..... 148<br/>           テキストのキル ( kill ) ..... 242         </p> <p>           初期化ファイル、readline [しょきかファイル、readline] ..... 244         </p> <p>           コメント ..... 15<br/>           クォート ..... 16<br/>           キル・リング ..... 242<br/>           テキストのヤंक ( yank ) [テキストのヤंक] ..... 242<br/>           キャッチポイント ..... 38<br/>           カレント・スレッド ..... 26<br/>           ウォッチポイントとスレッド ..... 37<br/>           ブレイクポイントの範囲指定 [ブレイクポイントのはんいしてい] ..... 32         </p> <p>           負のブレイクポイント番号 [ふのブレイクポイントばんごう] ..... 35         </p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- 設定、ウォッチポイントの [せってい、  
ウォッチポイントの] ..... 36
- キャッチポイント ..... 31
- ブレイクポイント、メモリ・アドレスの [ブ  
レイクポイント、メモリ・アドレスの]  
..... 31
- メモリ・アドレスのブレイクポイント [メモ  
リ・アドレスのブレイクポイント] .. 31
- ウォッチポイント ..... 31
- イベントに対するブレイクポイント [イベ  
ントにたいするブレイクポイント] .. 31
- ブレイクポイント、変数変化の [ブレイクポ  
イント、へんすうへんかの] ..... 31
- 変数変化のブレイクポイント [へんすうへ  
んかのブレイクポイント] ..... 31
- 自動的選択、スレッドの [じどうてきせんた  
く、スレッドの] ..... 28
- 番号、ブレイクポイント [ばんごう、ブレイ  
クポイント] ..... 31
- 最適化コードのデバッグ [さいてきかコー  
ドのデバッグ] ..... 21
- ウォッチポイントの設定 [ウォッチポイント  
のせってい] ..... 36
- 最後のブレイクポイント [さいごのブレイ  
クポイント] ..... 32
- スレッドの自動的選択 [スレッドのじどうて  
きせんたく] ..... 28
- ブレイクポイント番号 [ブレイクポイント  
ばんごう] ..... 31
- 繰り返し、コマンドの [くりかえし、コマン  
ドの] ..... 15
- 切り替え、スレッドの [きりかえ、スレッド  
の] ..... 26
- 補完、引用文字列の [ほかん、いんようもじ  
れつの] ..... 16
- 引用、コマンド内の [いんよう、コマンドな  
いの] ..... 16
- コマンドの繰り返し [コマンドのくりかえ  
し] ..... 15
- スレッドの切り替え [スレッドのきりかえ]  
..... 26
- 引用文字列の補完 [いんようもじれつのほ  
かん] ..... 16
- 対象、デバッグの [たいしょう、デバッグ  
の] ..... 26
- コマンド内の引用 [コマンドないのいんよ  
う] ..... 16

|                       |                        |
|-----------------------|------------------------|
| 再開、スレッドの [さいかい、スレッドの] | 最下位のフレーム [さいかいのフレーム]   |
| ..... 50              | ..... 51               |
| ハンドラ、例外の [ハンドラ、れいがいの] | 呼び出しスタック [よびだしスタック] .. |
| ..... 55              | ..... 51               |
| トレースバック .....         | オーバーロード .....          |
| 52                    | 44                     |
| スタックトレース .....        | オーバーロード、C++での .....    |
| 52                    | 88                     |
| スタック・フレーム .....       | シンボルの表現、C++の [シンボルのひょう |
| 51                    | げん、C++の] .....         |
| カレントなスタック・フレーム .....  | 73                     |
| 52                    | スレッド識別子、GDB の [スレッドしきべ |
| フレームを持たない関数の実行 [フレーム  | つし、GDB の] .....        |
| をもたないかんすうのじっこう]....   | 27                     |
| 51                    |                        |
| 表示を伴わないフレーム選択 [ひょうじを  | 変更、変数値の [へんこう、へんすうちの]  |
| ともなわないフレームせんたく]....   | ..... 105              |
| 52                    |                        |
| 選択されたフレーム [せんたくされたフ   | 註釈、割り込み [ちゅうしゃく、わりこみ]  |
| レーム] .....            | ..... 176              |
| 51                    | 註釈、フレーム [ちゅうしゃく、フレーム]  |
|                       | ..... 173              |
| 最上位のフレーム [さいじょういのフレー  |                        |
| ム] .....              |                        |
| 51                    |                        |
| スレッド、実行の [スレッド、じっこうの] | 設定、変数値の [せってい、へんすうちの]  |
| ..... 26              | ..... 105              |
| キャッチ、例外の [キャッチ、れいがいの] |                        |
| ..... 55              |                        |
| 停止したスレッド [ていししたスレッド]  | 出力、データの [しゅつりょく、データの]  |
| ..... 49              | ..... 63               |
| 処理、シグナルの [しより、シグナルの]  | 更新、変数値の [こうしん、へんすうちの]  |
| ..... 48              | ..... 105              |

- 割り込み、註釈 [わりこみ、ちゅうしゃく] ..... 176
- フレーム、註釈 [フレーム、ちゅうしゃく] ..... 173
- 戻る、関数から [もどる、かんすうから] ..... 107
- 返る、関数から [かえる、かんすうから] ..... 107
- 共有ライブラリ [きょうゆうライブラリ] ..... 112
- デバッグの対象 [デバッグのたいしょう] ..... 26
- 表示、データの [ひょうじ、データの] .. 63
- 調査、メモリの [ちょうさ、メモリの] .. 67
- 致命的シグナル [ちめいてきシグナル] .. 48, 237
- 多重ターゲット [たじゅうターゲット] .. 115
- 実行のスレッド [じっこうのスレッド] .. 26
- 関数の呼び出し [かんすうのよびだし] .. 108
- プロセス、多重 [プロセス、たじゅう] .. 28
- フレームの番号 [フレームのばんごう] .. 51
- バージョン番号 [バージョンばんごう] .. 19
- スレッドの再開 [スレッドのさいかい] .. 50
- スコープ演算子 [スコープえんざんし] .. 93
- オンライン文書 [オンラインぶんしょ] .. 17
- 初期化ファイル [しょきかファイル] ... 165
- 検査、データの [けんさ、データの] ..... 63
- フレーム、定義 [フレーム、ていぎ] ..... 51
- チェック、範囲 [チェック、はんい] ..... 83
- シンボルの名前 [シンボルのなまえ] ... 101
- イベント指定子 [イベントしていし] ... 259
- 未知のアドレス [みちのアドレス] ..... 66
- チェックサム、GDB リモートの ..... 123
- 表記上の慣習、GDB/MI [ひょうきじょうのかんしゅう、GDB/MI] ..... 179

|                                                  |     |                         |     |
|--------------------------------------------------|-----|-------------------------|-----|
| 結果レコード、GDB/MI[けっかレコード、<br>GDB/MI] .....          | 183 | 註釈、エラー [ちゅうしゃく、エラー]..   | 176 |
| ダウンロード、H8/300 や H8/500 への ..                     | 145 | エラー、註釈 [エラー、ちゅうしゃく]..   | 176 |
| 組み込み機能、Modula-2 の [くみこみきの<br>う、Modula-2 の] ..... | 91  | 正しくない値 [ただしくないあたい]..... | 64  |
| ダウンロード、Nindy-960 への .....                        | 148 | コマンド編集 [コマンドへんしゅう] ...  | 241 |
| シミュレータ、Z8000 .....                               | 155 | 例外ハンドラ [れいがいハンドラ] ..... | 38  |
| 変数名の衝突 [へんすうめいのしょうとつ]<br>.....                   | 64  | 馬鹿げた質問 [ばかげたしつもん] ..... | 161 |
| 整形した出力 [せいけいしたしゅつりょく]<br>.....                   | 66  | 多重スレッド [たじゅうスレッド] ..... | 26  |
| 変数への代入 [へんすうへのだいにゅう]<br>.....                    | 105 | 実行ファイル [じっこうファイル] ..... | 109 |
| 入力、数値の [にゅうりょく、すうちの]<br>.....                    | 159 | 逆アセンブル [ぎゃくアセンブル] ..... | 60  |
| 停止、出力の [ていし、しゅつりょくの]<br>.....                    | 159 | 履歴ファイル [りれきファイル] .....  | 158 |
| 発生、例外の [はっせい、れいがいの] ..                           | 39  | 履歴イベント [りれきイベント] .....  | 259 |
|                                                  |     | 履歴のサイズ [りれきのサイズ] .....  | 158 |
|                                                  |     | 範囲チェック [はんいチェック] .....  | 83  |



- 通過カウント [つかかカウント] ..... 42
- 初期フレーム [しょきフレーム] ..... 51
- データの検査 [データのけんさ] ..... 63  
デフォルト、C/C++の ..... 88
- 非同期出力、GDB/MI[ひどうきしゅつりょく、GDB/MI] ..... 181
- プロンプト ..... 157  
コマンド編集 [コマンドへんしゅう] ... 157  
サイズ、画面 [サイズ、がめん] ..... 159  
プロトコル、GDB リモート・シリアル ..... 123  
データ操作、GDB/MI[データそうさ、GDB/MI] ..... 192
- 浮動小数点、MIPS リモートの [ふどうしょうすうてん、MIPS リモートの] ... 151
- デフォルト、Modula-2 の ..... 93  
コンパイル、Sparclet ..... 153
- 表示、註釈 [ひょうじ、ちゅうしゃく] .. 175
- 註釈、表示 [ちゅうしゃく、ひょうじ] .. 175  
註釈、警告 [ちゅうしゃく、けいこく] .. 176
- 警告、註釈 [けいこく、ちゅうしゃく] .. 176
- 不正な入力 [ふせいなにゅうりょく] ... 237
- 慎重な動作 [しんちょうなどうさ] ..... 161
- 履歴の記録 [りれきのきろく] ..... 158
- 命令セット [めいれいセット] ..... 61
- 動的リンク [どうてきリンク] ..... 111
- メンバ関数 [メンバかんすう] ..... 87  
バグの報告 [バグのほうこく] ..... 237  
バグの基準 [バグのきじゅん] ..... 237
- 型チェック [かたチェック] ..... 82
- 画面サイズ [がめんサイズ] ..... 159
- スタック、Alpha の ..... 156  
チェック、C/C++の ..... 88  
バグ報告、GDB の [バグほうこく、GDB の] ..... 237

|                                          |     |                                                                           |     |
|------------------------------------------|-----|---------------------------------------------------------------------------|-----|
| 入力構文、GDB/MI[にゅうりょくこうぶん、<br>GDB/MI] ..... | 179 | 応答時間、MIPS デバッグの [おうとうじか<br>ん、MIPS デバッグの] .....                            | 156 |
| 出力構文、GDB/MI[しゅつりょくこうぶん、<br>GDB/MI] ..... | 180 | スタック、MIPS の .....                                                         | 156 |
| 通知出力、GDB/MI[つうちしゅつりょく、<br>GDB/MI] .....  | 181 | チェック、Modula-2 の .....                                                     | 93  |
| ログ出力、GDB/MI[ログしゅつりょく、<br>GDB/MI] .....   | 182 | 差異、標準 Modula-2 との [さい、ひょう<br>じゅん Modula-2 との] .....                       | 93  |
| シグナル .....                               | 48  | コマンド、STDEBUG ( ST2000 ) に対する<br>[コマンド、STDEBUG ( ST2000 ) にたい<br>する] ..... | 155 |
| シグナルの処理 [シグナルのしやり] .....                 | 48  | パケット、stdout への出力 [パケット、<br>stdout へのしゅつりょく] .....                         | 161 |
| ステップ実行 [ステップじっこう] .....                  | 45  | アタッチ .....                                                                | 25  |
| 例外の発生 [れいがいはっせい] .....                   | 39  | 説明文字列 [せつめいもじれつ] .....                                                    | 17  |
| 実行の再開 [じっこうのさいかい] .....                  | 45  | ヘルプ情報 [ヘルプじょうほう] .....                                                    | 17  |
| コマンド、C++専用の [コマンド、C++せん<br>ようの] .....    | 88  | 単語の補完 [たんごのほかん] .....                                                     | 15  |
| 名前空間、C++の [なまえくうかん、C++の]<br>.....        | 87  | 呼び出し、makeの [よびだし、make の] ..                                               | 13  |
| シーケンス ID、GDB リモートの .....                 | 123 | 値履歴と\$_や\$__[あたいりれきと\$_や\$__]<br>.....                                    | 68  |
|                                          |     | 註釈、値 [ちゅうしゃく、あたい] .....                                                   | 172 |
|                                          |     | 値、註釈 [あたい、ちゅうしゃく] .....                                                   | 172 |

- 参照宣言 [さんしょうせんげん] ..... 87
- 表示設定 [ひょうじせってい] ..... 70
- 正規表現 [せいきひょうげん] ..... 34
- 制御端末 [せいぎょたんまつ] ..... 24
- 数値表現 [すうちひょうげん] ..... 159
- 人工配列 [じんこうはいれつ] ..... 65
- 継続実行 [けいぞくじっこう] ..... 45
- 例外処理 [れいがいしゅり] ..... 38
- 履歴番号 [りれきばんごう] ..... 75
- 履歴展開 [りれきてんかい] ..... 158, 259
- 式の表示 [しきのひょうじ] ..... 68
- 自動表示 [じどうひょうじ] ..... 68
- 作業言語 [さぎょうげんご] ..... 79
- 履歴置換 [りれきちかん] ..... 158
- 割り込み [わりこみ] ..... 13
- 演算子、C/C++の [えんざんし、C/C++の] ..... 84
- 型変換、C++での [かたへんかん、C++での] ..... 87
- 識別子、GDB のスレッド [しきべつし、GDB のスレッド] ..... 27
- 互換性、GDB/MI と CLI [ごかんせい、GDB/MI と CLI] ..... 183
- 演算子、Modula-2 の [えんざんし、Modula-2 の] ..... 90
- 表記法、readline [ひょうきほう、readline] ..... 241
- 入出力 [にゅうしゅつりょく] ..... 24
- 省略形 [しょうりゃくけい] ..... 15
- 短縮形 [たんしゅくけい] ..... 15
- 値履歴 [あたいりれき] ..... 75
- 行指定 [ぎょうしてい] ..... 57

|                                                          |     |                   |     |
|----------------------------------------------------------|-----|-------------------|-----|
| 富士通 [ふじつう] .....                                         | 119 | 略称 [りやくしょう] ..... | 15  |
| 定数、C/C++の [ていすう、C/C++の] ...                              | 86  | 註釈 [ちゅうしゃく] ..... | 171 |
| 報告、GDB のバグ [ほうこく、GDB のバグ] .....                          | 237 | 編集 [へんしゅう] .....  | 157 |
| 終了、GDB の [しゅうりょう、GDB の] ..                               | 13  | 代入 [だいにゅう] .....  | 105 |
| 定数、Modula-2 の [ていすう、Modula-2 の] .....                    | 92  | 継承 [けいしょう] .....  | 89  |
| 操作、readline[そうさ、readline] .....                          | 241 | 端末 [たんまつ] .....   | 24  |
| 註釈、server 接頭語 [ちゅうしゃく、server<br>せつとうご] .....             | 171 | 実行 [じっこう] .....   | 21  |
| 実行、Sparclet[じっこう、Sparclet] ....                          | 153 | 検索 [けんさく] .....   | 58  |
| 変数/関数のコンテキスト、::[へんすう/かんすうのコンテキスト、::] .....               | 64  | 確認 [かくにん] .....   | 161 |
| 変数/関数のコンテキスト、2 つのコロン [へんすう/かんすうのコンテキスト、2<br>つのコロン] ..... | 64  | 引用 [いんよう] .....   | 101 |
| 変数/関数のコンテキスト、2 重コロン [へんすう/かんすうのコンテキスト、2<br>じゅうコロン] ..... | 64  | 補完 [ほかん] .....    | 15  |
|                                                          |     | 日立 [ひたち] .....    | 119 |

|                                        |     |                                   |     |
|----------------------------------------|-----|-----------------------------------|-----|
| 言語 [げんご].....                          | 79  | --annotate .....                  | 12  |
| 起動 [きどう].....                          | 21  | --async.....                      | 12  |
| 開始 [かいし].....                          | 21  | --batch.....                      | 11  |
| 式、C/C++の [しき、C/C++の].....              | 84  | --baud.....                       | 12  |
| 式、C++の [しき、C++の].....                  | 86  | --cd.....                         | 12  |
| 式、Modula-2 での [しき、Modula-2 での].....    | 90  | --command.....                    | 10  |
| 日立 SH へのダウンロード [ひたち SH へのダウンロード].....  | 145 | --core.....                       | 10  |
| 式 [しき].....                            | 63  | --directory .....                 | 10  |
| #                                      |     | --epoch.....                      | 12  |
| # (a comment).....                     | 15  | --exec.....                       | 10  |
| \$                                     |     | --fullname .....                  | 12  |
| \$ .....                               | 75  | --interpreter .....               | 13  |
| \$\$.....                              | 75  | --mapped.....                     | 10  |
| \$_と info breakpoints .....            | 34  | --noasync.....                    | 12  |
| \$_と info line.....                    | 60  | --nowindows .....                 | 11  |
| \$_, convenience variable.....         | 76  | --nx.....                         | 11  |
| \$_, convenience variable.....         | 76  | --quiet.....                      | 11  |
| \$_exitcode, convenience variable..... | 76  | --readnow.....                    | 11  |
| \$bpnum, convenience variable.....     | 32  | --se.....                         | 10  |
| \$cdir, convenience variable.....      | 59  | --silent.....                     | 11  |
| \$cwd, convenience variable.....       | 59  | --statistics .....                | 13  |
|                                        |     | --symbols.....                    | 10  |
|                                        |     | --tty.....                        | 12  |
|                                        |     | --version.....                    | 13  |
|                                        |     | --windows.....                    | 12  |
|                                        |     | --write.....                      | 13  |
|                                        |     | -b.....                           | 12  |
|                                        |     | -break-after .....                | 185 |
|                                        |     | -break-condition.....             | 185 |
|                                        |     | -break-delete .....               | 186 |
|                                        |     | -break-disable .....              | 186 |
|                                        |     | -break-enable .....               | 187 |
|                                        |     | -break-info .....                 | 187 |
|                                        |     | -break-insert .....               | 188 |
|                                        |     | -break-list .....                 | 189 |
|                                        |     | -break-watch.....                 | 190 |
|                                        |     | -c.....                           | 10  |
|                                        |     | -d.....                           | 10  |
|                                        |     | -data-disassemble.....            | 192 |
|                                        |     | -data-evaluate-expression .....   | 194 |
|                                        |     | -data-list-changed-registers..... | 195 |
|                                        |     | -data-list-register-names .....   | 195 |
|                                        |     | -data-list-register-values .....  | 196 |
|                                        |     | -data-read-memory.....            | 198 |
|                                        |     | -display-delete.....              | 199 |
|                                        |     | -display-disable.....             | 200 |



|                                                       |     |
|-------------------------------------------------------|-----|
| :                                                     |     |
| ::、変数/関数のコンテキスト [::、へんすう/かんすうのコンテキスト].....            | 64  |
| ::, in Modula-2 .....                                 | 93  |
| @                                                     |     |
| @, referencing memory as an array.....                | 65  |
| ,                                                     |     |
| “No symbol "foo" in current context” ....             | 65  |
| {                                                     |     |
| {type} .....                                          | 63  |
| ^                                                     |     |
| ^done .....                                           | 183 |
| ^error .....                                          | 183 |
| ^running .....                                        | 183 |
| 2                                                     |     |
| 2 つのコロン、変数/関数のコンテキスト [2 つのコロン、へんすう/かんすうのコンテキスト] ..... | 64  |
| 2 重コロン、変数/関数のコンテキスト [2 じゅうコロン、へんすう/かんすうのコンテキスト] ..... | 64  |
| 29K プログラムの実行 [29K プログラムのじっこう] .....                   | 142 |

## A

|                                     |     |
|-------------------------------------|-----|
| abort (C-g) .....                   | 257 |
| accept-line (Newline, Return) ..... | 252 |
| add-shared-symbol-file .....        | 112 |
| add-symbol-file .....               | 111 |
| Alpha のスタック .....                   | 156 |
| AMD EB29K .....                     | 142 |
| AMD29K レジスタ・スタック .....              | 156 |
| apropos .....                       | 18  |
| arg-begin .....                     | 172 |
| arg-end .....                       | 172 |
| arg-name-end .....                  | 172 |

|                                          |     |
|------------------------------------------|-----|
| arg-value .....                          | 172 |
| array-section-end .....                  | 173 |
| AT&T 仕様の逆アセンブル [AT&T しょうのぎゃくアセンブル] ..... | 61  |
| attach .....                             | 25  |
| awatch .....                             | 36  |

## B

|                                        |     |
|----------------------------------------|-----|
| b (break) .....                        | 32  |
| backtrace .....                        | 52  |
| backward-char (C-b) .....              | 252 |
| backward-delete-char (Rubout) .....    | 254 |
| backward-kill-line (C-x Rubout) .....  | 254 |
| backward-kill-word (M-DEL) .....       | 255 |
| backward-word (M-b) .....              | 252 |
| beginning-of-history (M-<) .....       | 253 |
| beginning-of-line (C-a) .....          | 252 |
| bell-style .....                       | 244 |
| break .....                            | 32  |
| break ... thread <i>threadno</i> ..... | 49  |
| breakpoint .....                       | 178 |
| breakpoint サブルーチン、リモート ...             | 120 |
| breakpoints-headers .....              | 176 |
| breakpoints-invalid .....              | 177 |
| breakpoints-table .....                | 176 |
| breakpoints-table-end .....            | 177 |
| bt (backtrace) .....                   | 52  |

## C

|                                       |    |
|---------------------------------------|----|
| c (continue) .....                    | 45 |
| C/C++ .....                           | 84 |
| C++ .....                             | 84 |
| C++とオブジェクトの形式 [C++とオブジェクトのけいしき] ..... | 86 |
| C++のシンボルの表現 [C++のシンボルのひょうげん] .....    | 73 |
| C++のスコープ解決 [C++のスコープかいけつ] .....       | 64 |
| C++のシンボル表示 [C++のシンボルひょうじ] .....       | 89 |
| C++のサポート、COFF でない .....               | 86 |
| C++の例外処理 [C++のれいがいしゅり] ..             | 88 |
| C++と ECOFF .....                      | 86 |
| C++と ELF/DWARF .....                  | 86 |
| C++と ELF/stabs .....                  | 86 |
| C++と XCOFF .....                      | 86 |

|                                                             |         |
|-------------------------------------------------------------|---------|
| call.....                                                   | 108     |
| call-last-kbd-macro (C-x e).....                            | 257     |
| calling make.....                                           | 13      |
| capitalize-word (M-c).....                                  | 254     |
| catch.....                                                  | 38      |
| catch catch.....                                            | 38      |
| catch exec.....                                             | 38      |
| catch fork.....                                             | 38      |
| catch load.....                                             | 38      |
| catch throw.....                                            | 38      |
| catch unload.....                                           | 38      |
| catch vfork.....                                            | 38      |
| cd.....                                                     | 24      |
| cdir.....                                                   | 59      |
| character-search (C-]).....                                 | 258     |
| character-search-backward (M-C-])...                        | 258     |
| Chill.....                                                  | 1       |
| clear.....                                                  | 39      |
| clear-screen (C-l).....                                     | 252     |
| COFF と C++.....                                             | 86      |
| commands.....                                               | 43, 175 |
| comment-begin.....                                          | 244     |
| complete.....                                               | 18      |
| complete (TAB).....                                         | 256     |
| completion-query-items.....                                 | 244     |
| condition.....                                              | 41      |
| configureによる構成、GDBの [configure<br>による構成、GDBの].....          | 263     |
| connect ( STDBUG に対する ) [connect<br>( STDBUG にたいする ) ]..... | 155     |
| continue.....                                               | 45      |
| convert-meta.....                                           | 245     |
| copy-backward-word ().....                                  | 255     |
| copy-forward-word ().....                                   | 255     |
| copy-region-as-kill ().....                                 | 255     |
| core.....                                                   | 111     |
| core-file.....                                              | 111     |
| Ctrl-C、リモート・デバッグ処理 [Ctrl-C、<br>リモート・デバッグしより].....           | 120     |
| cwd.....                                                    | 59      |

## D

|                         |     |
|-------------------------|-----|
| d (delete).....         | 39  |
| define.....             | 163 |
| delete.....             | 39  |
| delete breakpoints..... | 39  |
| delete display.....     | 69  |

|                                                   |     |
|---------------------------------------------------|-----|
| delete-char (C-d).....                            | 253 |
| delete-char-or-list ().....                       | 256 |
| delete-horizontal-space ().....                   | 255 |
| detach.....                                       | 25  |
| device.....                                       | 145 |
| digit-argument (M-0, M-1, ... M--)...             | 256 |
| dir.....                                          | 59  |
| directory.....                                    | 59  |
| dis (disable).....                                | 40  |
| disable.....                                      | 40  |
| disable breakpoints.....                          | 40  |
| disable display.....                              | 69  |
| disable-completion.....                           | 245 |
| disassemble.....                                  | 60  |
| display.....                                      | 69  |
| display-begin.....                                | 175 |
| display-end.....                                  | 175 |
| display-expression.....                           | 175 |
| display-expression-end.....                       | 175 |
| display-format.....                               | 175 |
| display-number-end.....                           | 175 |
| display-value.....                                | 175 |
| do (down).....                                    | 53  |
| do-uppercase-version (M-a, M-b, M-x,<br>...)..... | 257 |
| document.....                                     | 163 |
| down.....                                         | 53  |
| down-silently.....                                | 54  |
| downcase-word (M-l).....                          | 254 |
| dump-functions ().....                            | 258 |
| dump-macros ().....                               | 258 |
| dump-variables ().....                            | 258 |

## E

|                                            |     |
|--------------------------------------------|-----|
| 'eb.log'.....                              | 144 |
| EB29K 用のログ・ファイル [EB29K 用の<br>ログ・ファイル]..... | 144 |
| EB29K ボード.....                             | 142 |
| EBMON.....                                 | 143 |
| echo.....                                  | 165 |
| editing-mode.....                          | 245 |
| else.....                                  | 163 |
| elt.....                                   | 173 |
| elt-rep.....                               | 173 |
| elt-rep-end.....                           | 173 |
| Emacs.....                                 | 167 |
| enable.....                                | 40  |



enable breakpoints ..... 40  
 enable display ..... 69  
 enable-keypad ..... 245  
 end ..... 43  
 end-kbd-macro (C-x ) ..... 257  
 end-of-history (M->) ..... 253  
 end-of-line (C-e) ..... 252  
 error ..... 176  
 error-begin ..... 176  
 exceptionHandler ..... 121  
 exchange-point-and-mark (C-x C-x) ... 257  
 exec-file ..... 109  
 exited ..... 177  
 expand-tilde ..... 245

## F

f (frame) ..... 53  
 fg (resume foreground execution) ..... 45  
 field ..... 176  
 field-begin ..... 172  
 field-end ..... 172  
 field-name-end ..... 172  
 field-value ..... 172  
 file ..... 109  
 finish ..... 46  
 flush\_i\_cache ..... 121  
 foo ..... 114  
 forkを呼び出す関数のデバッグ [fork をよびだすかんすうのデバッグ] ..... 28  
 Fortran ..... 1  
 forward-backward-delete-char () ..... 254  
 forward-char (C-f) ..... 252  
 forward-search ..... 58  
 forward-search-history (C-s) ..... 253  
 forward-word (M-f) ..... 252  
 frame, command ..... 52  
 frame, selecting ..... 53  
 frame-address ..... 174  
 frame-address-end ..... 174  
 frame-args ..... 174  
 frame-begin ..... 173  
 frame-end ..... 173  
 frame-function-name ..... 174  
 frame-source-begin ..... 174  
 frame-source-end ..... 174  
 frame-source-file ..... 174  
 frame-source-file-end ..... 174

frame-source-line ..... 174  
 frame-where ..... 174  
 frames-invalid ..... 177  
 Fujitsu ..... 119  
 function-call ..... 173

## G

g++, GNU C++コンパイラ ..... 84  
 GDB リファレンス・カード ..... 261  
 GDB のスレッド識別子 [GDB のスレッドしきべつし] ..... 27  
 GDB のバグの報告 [GDB のバグのほうこく] ..... 237  
 GDB の終了 [GDB のしゅうりょう] ..... 13  
 'gdb.ini' ..... 165  
 GDB/MI、ブレイクポイント・コマンド [GDB/MI、ブレイクポイント・コマンド] ..... 185  
 GDB/MI、ストリーム・レコード [GDB/MI、ストリーム・レコード] ..... 183  
 GDB/MI、出力構文の変更案 [GDB/MI、しゅつりょくこうぶんのへんこうあん] ..... 233  
 GDB/MI、変数オブジェクト [GDB/MI、へんすうオブジェクト] ..... 230  
 GDB/MI、ターゲット出力 [GDB/MI、ターゲットしゅつりょく] ..... 182  
 GDB/MI、ステータス出力 [GDB/MI、ステータスしゅつりょく] ..... 181  
 GDB/MI、コンソール出力 [GDB/MI、コンソールしゅつりょく] ..... 181  
 GDB/MI、帯域外レコード [GDB/MI、たいいきがいレコード] ..... 184  
 GDB/MI、表記上の慣習 [GDB/MI、ひょうきじょうのかんしゅう] ..... 179  
 GDB/MI、結果レコード [GDB/MI、けっかレコード] ..... 183  
 GDB/MI、非同期出力 [GDB/MI、ひどうきしゅつりょく] ..... 181  
 GDB/MI、データ操作 [GDB/MI、データそうさ] ..... 192  
 GDB/MI、入力構文 [GDB/MI、にゅうりょくこうぶん] ..... 179  
 GDB/MI、出力構文 [GDB/MI、しゅつりょくこうぶん] ..... 180  
 GDB/MI、通知出力 [GDB/MI、つうちしゅつりょく] ..... 181

|                                            |     |
|--------------------------------------------|-----|
| GDB/MI、ログ出力 [GDB/MI、ログしゅつりょく]              | 182 |
| GDB/MI、簡単な例 [GDB/MI、かんたんなれい]               | 182 |
| GDB/MI、目的 [GDB/MI、もくてき]                    | 179 |
| GDB/MI、CLI に対する互換性 [GDB/MI、CLI にたいするごかんせい] | 183 |
| GDBHISTFILE                                | 158 |
| gdbserve.nlm                               | 135 |
| gdbserver                                  | 134 |
| getDebugChar                               | 120 |
| GNU C++                                    | 84  |
| GNU Emacs                                  | 167 |

## H

|                                    |     |
|------------------------------------|-----|
| h (help)                           | 17  |
| H8/300 や H8/500 へのダウンロード           | 145 |
| handle                             | 48  |
| handle_exception                   | 119 |
| hbreak                             | 33  |
| help                               | 17  |
| help target                        | 116 |
| help user-defined                  | 163 |
| heuristic-fence-post (Alpha, MIPS) | 156 |
| history-search-backward ()         | 253 |
| history-search-forward ()          | 253 |
| Hitachi                            | 119 |
| horizontal-scroll-mode             | 245 |
| HP-UX、スレッド識別子 (システム)               |     |
| [HP-UX、スレッドしきべつし (システム)]           | 27  |
| HP-UX における New <i>syntag</i> メッセージ | 27  |

## I

|                    |     |
|--------------------|-----|
| i (info)           | 18  |
| i386               | 119 |
| 'i386-stub.c'      | 119 |
| i960               | 148 |
| if                 | 163 |
| ignore             | 42  |
| INCLUDE_RDB        | 140 |
| info               | 18  |
| info address       | 101 |
| info all-registers | 77  |
| info args          | 54  |
| info breakpoints   | 34  |

|                                       |     |
|---------------------------------------|-----|
| info catch                            | 55  |
| info display                          | 69  |
| info extensions                       | 81  |
| info f (info frame)                   | 54  |
| info files                            | 112 |
| info float                            | 78  |
| info frame                            | 54  |
| info frame, show the source language  | 81  |
| info functions                        | 102 |
| info line                             | 60  |
| info locals                           | 55  |
| info proc                             | 139 |
| info proc id                          | 139 |
| info proc mappings                    | 139 |
| info proc status                      | 139 |
| info proc times                       | 139 |
| info program                          | 31  |
| info registers                        | 77  |
| info s (info stack)                   | 52  |
| info set                              | 19  |
| info share                            | 112 |
| info sharedlibrary                    | 112 |
| info signals                          | 48  |
| info source                           | 102 |
| info source, show the source language | 81  |
| info sources                          | 102 |
| info stack                            | 52  |
| info target                           | 112 |
| info terminal                         | 24  |
| info threads                          | 27  |
| info types                            | 102 |
| info variables                        | 102 |
| info watchpoints                      | 36  |
| input-meta                            | 245 |
| insert-comment (M-#)                  | 258 |
| insert-completions (M-*)              | 256 |
| inspect                               | 63  |
| Intel                                 | 119 |
| Intel 仕様の逆アセンブル [Intel しょうのぎゃくアセンブル]  | 61  |
| isearch-terminators                   | 245 |

## J

|      |     |
|------|-----|
| jump | 106 |
|------|-----|

## K

|                          |     |
|--------------------------|-----|
| keymap .....             | 245 |
| kill .....               | 26  |
| kill-line (C-k) .....    | 254 |
| kill-region () .....     | 255 |
| kill-whole-line () ..... | 255 |
| kill-word (M-d) .....    | 255 |
| KOD .....                | 136 |

## L

|                            |     |
|----------------------------|-----|
| l (list) .....             | 57  |
| list .....                 | 57  |
| load <i>filename</i> ..... | 117 |

## M

|                                                |     |
|------------------------------------------------|-----|
| m680x0 .....                                   | 119 |
| 'm68k-stub.c' .....                            | 119 |
| maint info breakpoints .....                   | 35  |
| maint print psymbols .....                     | 103 |
| maint print symbols .....                      | 103 |
| make .....                                     | 13  |
| makeの呼び出し [make のよびだし] .....                   | 13  |
| mapped .....                                   | 110 |
| mark-modified-lines .....                      | 246 |
| memset .....                                   | 121 |
| menu-complete () .....                         | 256 |
| meta-flag .....                                | 245 |
| MIPS のスタック .....                               | 156 |
| MIPS リモート浮動小数点 [MIPS リモート<br>ふどうしょうすうてん] ..... | 151 |
| MIPS ボード .....                                 | 150 |
| MIPS remotedebugプロトコル .....                    | 151 |
| Modula-2 .....                                 | 1   |
| Modula-2 の# .....                              | 94  |
| Modula-2、GDB サポート .....                        | 90  |
| Motorola 680x0 .....                           | 119 |

## N

|                              |     |
|------------------------------|-----|
| n (next) .....               | 46  |
| New systag メッセージ .....       | 26  |
| New systag メッセージ、HP-UX ..... | 27  |
| next .....                   | 46  |
| next-history (C-n) .....     | 252 |
| nexti .....                  | 48  |

|                                                       |     |
|-------------------------------------------------------|-----|
| ni (nexti) .....                                      | 48  |
| Nindy .....                                           | 148 |
| Nindy-960 へのダウンロード .....                              | 148 |
| non-incremental-forward-search-history<br>(M-n) ..... | 253 |
| non-incremental-reverse-search-history<br>(M-p) ..... | 253 |

## O

|                       |     |
|-----------------------|-----|
| output .....          | 166 |
| output-meta .....     | 246 |
| overload-choice ..... | 175 |

## P

|                                  |     |
|----------------------------------|-----|
| Pascal .....                     | 1   |
| path .....                       | 23  |
| possible-completions (M-?) ..... | 256 |
| post-commands .....              | 175 |
| post-overload-choice .....       | 175 |
| post-prompt .....                | 175 |
| post-prompt-for-continue .....   | 176 |
| post-query .....                 | 175 |
| pre-commands .....               | 175 |
| pre-overload-choice .....        | 175 |
| pre-prompt .....                 | 175 |
| pre-prompt-for-continue .....    | 176 |
| pre-query .....                  | 175 |
| prefix-meta (ESC) .....          | 257 |
| previous-history (C-p) .....     | 252 |
| print .....                      | 63  |
| printf .....                     | 166 |
| prompt .....                     | 175 |
| prompt-for-continue .....        | 176 |
| ptype .....                      | 101 |
| putDebugChar .....               | 120 |
| pwd .....                        | 24  |

## Q

|                                  |     |
|----------------------------------|-----|
| q (quit) .....                   | 13  |
| query .....                      | 175 |
| quit .....                       | 176 |
| quit [ <i>expression</i> ] ..... | 13  |
| quoted-insert (C-q, C-v) .....   | 254 |

## R

|                                     |     |
|-------------------------------------|-----|
| r (run) .....                       | 21  |
| rbreak .....                        | 34  |
| re-read-init-file (C-x C-r) .....   | 257 |
| readline .....                      | 157 |
| readnow .....                       | 110 |
| record .....                        | 176 |
| redraw-current-line () .....        | 252 |
| remote debugging .....              | 118 |
| remotedebug、MIPS プロトコル .....        | 151 |
| remotetimeout .....                 | 153 |
| reset .....                         | 149 |
| RET (repeat last command) .....     | 15  |
| retransmit-timeout、MIPS プロトコル ..... | 152 |
| return .....                        | 107 |
| reverse-search .....                | 58  |
| reverse-search-history (C-r) .....  | 253 |
| revert-line (M-r) .....             | 257 |
| run .....                           | 21  |
| rwatch .....                        | 36  |

## S

|                                               |     |
|-----------------------------------------------|-----|
| s (step) .....                                | 46  |
| search .....                                  | 58  |
| section .....                                 | 112 |
| select-frame .....                            | 52  |
| self-insert (a, b, A, 1, !, ...) .....        | 254 |
| server 接頭語、註釈 [server セットとご、<br>ちゅうしゃく] ..... | 171 |
| server prefix for annotations .....           | 171 |
| set .....                                     | 18  |
| set args .....                                | 23  |
| set auto-solib-add .....                      | 113 |
| set check range .....                         | 83  |
| set check type .....                          | 82  |
| set check, range .....                        | 83  |
| set check, type .....                         | 82  |
| set complaints .....                          | 160 |
| set confirm .....                             | 161 |
| set debug arch .....                          | 161 |
| set debug event .....                         | 161 |
| set debug expression .....                    | 161 |
| set debug overload .....                      | 161 |
| set debug remote .....                        | 161 |
| set debug serial .....                        | 162 |

|                                      |     |
|--------------------------------------|-----|
| set debug target .....               | 162 |
| set debug varobj .....               | 162 |
| set demangle-style .....             | 73  |
| set disassembly-flavor .....         | 61  |
| set editing .....                    | 157 |
| set endian auto .....                | 118 |
| set endian big .....                 | 117 |
| set endian little .....              | 117 |
| set environment .....                | 23  |
| set extension-language .....         | 81  |
| set follow-fork-mode .....           | 29  |
| set gnutarget .....                  | 116 |
| set height .....                     | 159 |
| set history expansion .....          | 158 |
| set history filename .....           | 158 |
| set history save .....               | 158 |
| set history size .....               | 158 |
| set input-radix .....                | 159 |
| set language .....                   | 80  |
| set listsize .....                   | 57  |
| set machine .....                    | 147 |
| set memory mod .....                 | 148 |
| set mipsfpu .....                    | 151 |
| set opaque-type-resolution .....     | 103 |
| set output-radix .....               | 160 |
| set overload-resolution .....        | 89  |
| set print address .....              | 70  |
| set print array .....                | 71  |
| set print asm-demangle .....         | 73  |
| set print demangle .....             | 73  |
| set print elements .....             | 72  |
| set print max-symbolic-offset .....  | 71  |
| set print null-stop .....            | 72  |
| set print object .....               | 74  |
| set print pretty .....               | 72  |
| set print sevenbit-strings .....     | 72  |
| set print static-members .....       | 74  |
| set print symbol-filename .....      | 70  |
| set print union .....                | 72  |
| set print vtbl .....                 | 74  |
| set processor args .....             | 151 |
| set prompt .....                     | 157 |
| set remotedebug, MIPS protocol ..... | 151 |
| set retransmit-timeout .....         | 152 |
| set rstack_high_address .....        | 156 |
| set symbol-reloading .....           | 103 |
| set timeout .....                    | 152 |
| set variable .....                   | 105 |

|                                  |     |                                      |          |
|----------------------------------|-----|--------------------------------------|----------|
| set verbose.....                 | 160 | show print max-symbolic-offset.....  | 71       |
| set width.....                   | 159 | show print object.....               | 74       |
| set write.....                   | 108 | show print pretty.....               | 72       |
| set-mark (C-@).....              | 257 | show print sevenbit-strings.....     | 72       |
| set_debug_traps.....             | 119 | show print static-members.....       | 74       |
| SH.....                          | 119 | show print symbol-filename.....      | 71       |
| 'sh-stub.c'.....                 | 119 | show print union.....                | 73       |
| share.....                       | 112 | show print vtbl.....                 | 75       |
| sharedlibrary.....               | 112 | show processor.....                  | 151      |
| shell.....                       | 13  | show prompt.....                     | 157      |
| shell escape.....                | 13  | show remotedebug, MIPS protocol..... | 151      |
| show.....                        | 19  | show retransmit-timeout.....         | 152      |
| show args.....                   | 23  | show rstack_high_address.....        | 156      |
| show auto-solib-add.....         | 113 | show symbol-reloading.....           | 103      |
| show check range.....            | 83  | show timeout.....                    | 152      |
| show check type.....             | 82  | show user.....                       | 164      |
| show complaints.....             | 160 | show values.....                     | 75       |
| show confirm.....                | 161 | show verbose.....                    | 160      |
| show convenience.....            | 76  | show version.....                    | 19       |
| show copying.....                | 19  | show warranty.....                   | 19       |
| show debug arch.....             | 161 | show width.....                      | 159      |
| show debug event.....            | 161 | show write.....                      | 108      |
| show debug expression.....       | 161 | show-all-if-ambiguous.....           | 246      |
| show debug overload.....         | 161 | shows.....                           | 158      |
| show debug remote.....           | 162 | si (stepi).....                      | 47       |
| show debug serial.....           | 162 | signal.....                          | 107, 178 |
| show debug target.....           | 162 | signal-handler-caller.....           | 173      |
| show debug varobj.....           | 162 | signal-name.....                     | 177      |
| show demangle-style.....         | 74  | signal-name-end.....                 | 177      |
| show directories.....            | 60  | signal-string.....                   | 177      |
| show editing.....                | 157 | signal-string-end.....               | 177      |
| show environment.....            | 23  | signalled.....                       | 177      |
| show gnutarget.....              | 116 | silent.....                          | 43       |
| show height.....                 | 159 | sim.....                             | 155      |
| show history.....                | 158 | source.....                          | 165, 178 |
| show input-radix.....            | 160 | Sparc.....                           | 119      |
| show language.....               | 81  | 'sparc-stub.c'.....                  | 119      |
| show listsize.....               | 57  | 'sparcl-stub.c'.....                 | 119      |
| show machine.....                | 147 | Sparclet.....                        | 153      |
| show mipsfpu.....                | 151 | Sparclet プログラムの実行とデバッグ               |          |
| show opaque-type-resolution..... | 103 | [Sparclet プログラムのじっこうとデ               |          |
| show output-radix.....           | 160 | バッグ].....                            | 154      |
| show paths.....                  | 23  | Sparclet へのダウンロード.....               | 154      |
| show print address.....          | 70  | Sparclet、コンパイル.....                  | 153      |
| show print array.....            | 71  | Sparclet、実行 [Sparclet、じっこう]....      | 153      |
| show print asm-demangle.....     | 73  | SparcLite.....                       | 119      |
| show print demangle.....         | 73  | speed.....                           | 145      |
| show print elements.....         | 72  |                                      |          |

|                                                    |     |
|----------------------------------------------------|-----|
| ST2000 用の補助的なコマンド [ST2000 よう<br>のほじょてきなコマンド] ..... | 155 |
| st2000 <i>cmd</i> .....                            | 155 |
| start-kbd-macro (C-x () .....                      | 257 |
| starting .....                                     | 177 |
| STDEBUG コマンド ( ST2000 ) .....                      | 155 |
| step .....                                         | 46  |
| stepi .....                                        | 47  |
| stop, a pseudo-command .....                       | 164 |
| stopping .....                                     | 177 |
| symbol-file .....                                  | 109 |

## T

|                                       |     |
|---------------------------------------|-----|
| tab-insert (M-TAB) .....              | 254 |
| target .....                          | 115 |
| target abug .....                     | 149 |
| target adapt .....                    | 142 |
| target amd-eb .....                   | 142 |
| target array .....                    | 151 |
| target bug .....                      | 150 |
| target core .....                     | 116 |
| target cpu32bug .....                 | 149 |
| target dbug .....                     | 149 |
| target ddb <i>port</i> .....          | 150 |
| target dink32 .....                   | 152 |
| target e7000, with H8/300 .....       | 145 |
| target e7000, with Hitachi ICE .....  | 147 |
| target e7000, with Hitachi SH .....   | 152 |
| target es1800 .....                   | 150 |
| target est .....                      | 149 |
| target exec .....                     | 116 |
| target hms, and serial protocol ..... | 146 |
| target hms, with H8/300 .....         | 145 |
| target hms, with Hitachi SH .....     | 152 |
| target lsi <i>port</i> .....          | 151 |
| target m32r .....                     | 149 |
| target mips <i>port</i> .....         | 150 |
| target mon960 .....                   | 148 |
| target nindy .....                    | 148 |
| target nrom .....                     | 117 |
| target op50n .....                    | 152 |
| target pmon <i>port</i> .....         | 150 |
| target ppccbug .....                  | 152 |
| target ppccbug1 .....                 | 152 |
| target r3900 .....                    | 151 |
| target rdi .....                      | 145 |
| target rdp .....                      | 145 |

|                                                        |     |
|--------------------------------------------------------|-----|
| target remote .....                                    | 116 |
| target remote、シリアル回線 [target<br>remote、シリアルかいせん] ..... | 122 |
| target remote、TCP ポート .....                            | 122 |
| target rom68k .....                                    | 149 |
| target sds .....                                       | 152 |
| target sh3, with H8/300 .....                          | 145 |
| target sh3, with SH .....                              | 152 |
| target sh3e, with H8/300 .....                         | 145 |
| target sh3e, with SH .....                             | 152 |
| target sim .....                                       | 116 |
| target sim, with Z8000 .....                           | 155 |
| target sparclite .....                                 | 154 |
| target vxworks .....                                   | 140 |
| target w89k .....                                      | 152 |
| tbreak .....                                           | 33  |
| thbreak .....                                          | 33  |
| this, inside C++ member functions .....                | 87  |
| thread apply .....                                     | 28  |
| thread <i>threadno</i> .....                           | 28  |
| tilde-expand (M-~) .....                               | 257 |
| timeout、MIPS プロトコル .....                               | 152 |
| transpose-chars (C-t) .....                            | 254 |
| transpose-words (M-t) .....                            | 254 |
| tty .....                                              | 24  |

## U

|                                          |     |
|------------------------------------------|-----|
| u (until) .....                          | 46  |
| udi .....                                | 142 |
| UDI .....                                | 142 |
| UDI 経由の AMD29K[UDI けいゆの<br>AMD29K] ..... | 142 |
| undisplay .....                          | 69  |
| undo (C-_, C-x C-u) .....                | 257 |
| universal-argument () .....              | 256 |
| unix-line-discard (C-u) .....            | 255 |
| unix-word-rubout (C-w) .....             | 255 |
| unset environment .....                  | 24  |
| until .....                              | 46  |
| up .....                                 | 53  |
| up-silently .....                        | 54  |
| upcase-word (M-u) .....                  | 254 |

## V

|                                           |     |
|-------------------------------------------|-----|
| value-begin .....                         | 172 |
| value-end .....                           | 172 |
| value-history-begin.....                  | 172 |
| value-history-end.....                    | 172 |
| value-history-value.....                  | 172 |
| visible-stats .....                       | 246 |
| VxWorks.....                              | 140 |
| VxWorks へのダウンロード .....                    | 141 |
| VxWorks タスクの実行 [VxWorks タスクの<br>実行] ..... | 141 |
| vxworks-timeout .....                     | 140 |

## W

|                  |     |
|------------------|-----|
| watch.....       | 36  |
| watchpoint ..... | 178 |
| whatis.....      | 101 |

|            |     |
|------------|-----|
| where..... | 52  |
| while..... | 163 |

## X

|                                 |    |
|---------------------------------|----|
| x (examine memory) .....        | 67 |
| x(examine), and info line ..... | 60 |

## Y

|                                |     |
|--------------------------------|-----|
| yank (C-y) .....               | 255 |
| yank-last-arg (M-., M-_) ..... | 253 |
| yank-nth-arg (M-C-y) .....     | 253 |
| yank-pop (M-y) .....           | 255 |

## Z

|                          |     |
|--------------------------|-----|
| Z8000.....               | 155 |
| Zilog Z8000 シミュレータ ..... | 155 |

The body of this manual is set in  
cmr10 at 10.95pt,  
with headings in **cmb10 at 10.95pt**  
and examples in cmtt10 at 10.95pt.  
*cmti10 at 10.95pt*,  
**cmb10 at 10.95pt**, and  
*cmsl10 at 10.95pt*  
are used for emphasis.



## 目次

|                                      |           |
|--------------------------------------|-----------|
| <b>GDB の要約</b> .....                 | <b>1</b>  |
| フリー・ソフトウェア .....                     | 1         |
| GDB に貢献した人々 .....                    | 1         |
| <b>1 GDB セッションのサンプル</b> .....        | <b>5</b>  |
| <b>2 GDB の起動と終了</b> .....            | <b>9</b>  |
| 2.1 GDB の起動 .....                    | 9         |
| 2.1.1 ファイルの選択 .....                  | 10        |
| 2.1.2 モードの選択 .....                   | 11        |
| 2.2 GDB の終了 .....                    | 13        |
| 2.3 シェル・コマンド .....                   | 13        |
| <b>3 GDB コマンド</b> .....              | <b>15</b> |
| 3.1 コマンドの構文 .....                    | 15        |
| 3.2 コマンド名の補完 .....                   | 15        |
| 3.3 ヘルプの表示 .....                     | 17        |
| <b>4 GDB 配下でのプログラムの実行</b> .....      | <b>21</b> |
| 4.1 デバッグのためのコンパイル .....              | 21        |
| 4.2 ユーザ・プログラムの起動 .....               | 21        |
| 4.3 ユーザ・プログラムの引数 .....               | 22        |
| 4.4 ユーザ・プログラムの環境 .....               | 23        |
| 4.5 ユーザ・プログラムの作業ディレクトリ .....         | 24        |
| 4.6 ユーザ・プログラムの入出力 .....              | 24        |
| 4.7 既に実行中のプロセスのデバッグ .....            | 25        |
| 4.8 子プロセスの終了 .....                   | 26        |
| 4.9 マルチスレッド・プログラムのデバッグ .....         | 26        |
| 4.10 マルチプロセス・プログラムのデバッグ .....        | 28        |
| <b>5 停止と継続</b> .....                 | <b>31</b> |
| 5.1 ブ레이크ポイント、ウォッチポイント、キャッチポイント ..... | 31        |
| 5.1.1 ブ레이크ポイントの設定 .....              | 32        |
| 5.1.2 ウォッチポイントの設定 .....              | 36        |
| 5.1.3 キャッチポイントの設定 .....              | 38        |
| 5.1.4 ブ레이크ポイントの削除 .....              | 39        |
| 5.1.5 ブ레이크ポイントの無効化 .....             | 40        |
| 5.1.6 ブ레이크ポイントの成立条件 .....            | 41        |
| 5.1.7 ブ레이크ポイント・コマンド・リスト .....        | 43        |
| 5.1.8 ブ레이크ポイント・メニュー .....            | 44        |
| 5.1.9 挿入できないブ레이크ポイント .....           | 44        |

|          |                           |           |
|----------|---------------------------|-----------|
| 5.2      | 継続実行とステップ実行 .....         | 45        |
| 5.3      | シグナル .....                | 48        |
| 5.4      | マルチスレッド・プログラムの停止と起動 ..... | 49        |
| <b>6</b> | <b>スタックの検査 .....</b>      | <b>51</b> |
| 6.1      | スタック・フレーム .....           | 51        |
| 6.2      | バックトレース .....             | 52        |
| 6.3      | フレームの選択 .....             | 53        |
| 6.4      | フレームに関する情報 .....          | 54        |
| <b>7</b> | <b>ソース・ファイルの検査 .....</b>  | <b>57</b> |
| 7.1      | ソース行の表示 .....             | 57        |
| 7.2      | ソース・ファイル内の検索 .....        | 58        |
| 7.3      | ソース・ディレクトリの指定 .....       | 59        |
| 7.4      | ソースとマシン・コード .....         | 60        |
| <b>8</b> | <b>データの検査 .....</b>       | <b>63</b> |
| 8.1      | 式 .....                   | 63        |
| 8.2      | プログラム変数 .....             | 64        |
| 8.3      | 人工配列 .....                | 65        |
| 8.4      | 出力フォーマット .....            | 66        |
| 8.5      | メモリの調査 .....              | 67        |
| 8.6      | 自動表示 .....                | 68        |
| 8.7      | 表示設定 .....                | 70        |
| 8.8      | 値履歴 .....                 | 75        |
| 8.9      | コンビニエンス変数 .....           | 76        |
| 8.10     | レジスタ .....                | 77        |
| 8.11     | 浮動小数ハードウェア .....          | 78        |
| <b>9</b> | <b>異なる言語の使い方 .....</b>    | <b>79</b> |
| 9.1      | ソース言語の切り替え .....          | 79        |
| 9.1.1    | ファイル拡張子と言語のリスト .....      | 79        |
| 9.1.2    | 作業言語の設定 .....             | 80        |
| 9.1.3    | GDB によるソース言語の推定 .....     | 80        |
| 9.2      | 言語の表示 .....               | 81        |
| 9.3      | 型と範囲のチェック .....           | 81        |
| 9.3.1    | 型チェックの概要 .....            | 82        |
| 9.3.2    | 範囲チェックの概要 .....           | 83        |
| 9.4      | サポートされる言語 .....           | 84        |
| 9.4.1    | C/C++ .....               | 84        |
| 9.4.1.1  | C/C++演算子 .....            | 84        |
| 9.4.1.2  | C/C++定数 .....             | 86        |
| 9.4.1.3  | C++式 .....                | 86        |
| 9.4.1.4  | C/C++のデフォルト .....         | 88        |
| 9.4.1.5  | C/C++の型チェックと範囲チェック .....  | 88        |
| 9.4.1.6  | GDB と C .....             | 88        |
| 9.4.1.7  | C++用の GDB 機能 .....        | 88        |

|           |                              |            |
|-----------|------------------------------|------------|
| 9.4.2     | Modula-2 .....               | 90         |
| 9.4.2.1   | Modula-2 演算子 .....           | 90         |
| 9.4.2.2   | 組み込み関数と組み込みプロシージャ .....      | 91         |
| 9.4.2.3   | 定数 .....                     | 92         |
| 9.4.2.4   | Modula-2 デフォルト .....         | 93         |
| 9.4.2.5   | 標準 Modula-2 との差異 .....       | 93         |
| 9.4.2.6   | Modula-2 の型チェックと範囲チェック ..... | 93         |
| 9.4.2.7   | スコープ演算子::と .....             | 93         |
| 9.4.2.8   | GDB と Modula-2 .....         | 94         |
| 9.4.3     | Chill .....                  | 94         |
| 9.4.3.1   | モードの表示方法 .....               | 94         |
| 9.4.3.2   | ロケーションとそのアクセス .....          | 96         |
| 9.4.3.3   | 値と操作 .....                   | 97         |
| 9.4.3.4   | Chill の型チェックと範囲チェック .....    | 99         |
| 9.4.3.5   | Chill のデフォルト .....           | 100        |
| <b>10</b> | <b>シンボル・テーブルの検査 .....</b>    | <b>101</b> |
| <b>11</b> | <b>実行処理の変更 .....</b>         | <b>105</b> |
| 11.1      | 変数への代入 .....                 | 105        |
| 11.2      | 異なるアドレスにおける処理継続 .....        | 106        |
| 11.3      | ユーザ・プログラムへのシグナルの通知 .....     | 107        |
| 11.4      | 関数からの復帰 .....                | 107        |
| 11.5      | プログラム関数の呼び出し .....           | 108        |
| 11.6      | プログラムへのパッチ適用 .....           | 108        |
| <b>12</b> | <b>GDB ファイル .....</b>        | <b>109</b> |
| 12.1      | ファイルを指定するコマンド .....          | 109        |
| 12.2      | シンボル・ファイル読み込み時のエラー .....     | 113        |
| <b>13</b> | <b>デバッグ・ターゲットの指定 .....</b>   | <b>115</b> |
| 13.1      | アクティブ・ターゲット .....            | 115        |
| 13.2      | ターゲットを管理するコマンド .....         | 115        |
| 13.3      | ターゲットのバイト・オーダの選択 .....       | 117        |
| 13.4      | リモート・デバッグ .....              | 118        |
| 13.4.1    | GDB リモート・シリアル・プロトコル .....    | 118        |
| 13.4.1.1  | スタブの提供する機能 .....             | 119        |
| 13.4.1.2  | スタブに対する必須作業 .....            | 120        |
| 13.4.1.3  | ここまでのまとめ .....               | 121        |
| 13.4.1.4  | 通信プロトコル .....                | 123        |
| 13.4.1.5  | gdbserverプログラムの使用 .....      | 134        |
| 13.4.1.6  | gdbserve.nlmプログラムの使用 .....   | 135        |
| 13.5      | カーネル・オブジェクト表示 .....          | 136        |

|           |                           |            |
|-----------|---------------------------|------------|
| <b>14</b> | <b>コンフィギュレーション固有の情報</b>   | <b>139</b> |
| 14.1      | ネイティブ                     | 139        |
| 14.1.1    | HP-UX                     | 139        |
| 14.1.2    | SVR4 プロセス情報               | 139        |
| 14.2      | 組み込みオペレーティング・システム         | 140        |
| 14.2.1    | VxWorks における GDB の使用      | 140        |
| 14.2.1.1  | VxWorks への接続              | 141        |
| 14.2.1.2  | VxWorks ダウンロード            | 141        |
| 14.2.1.3  | タスクの実行                    | 141        |
| 14.3      | 組み込みプロセッサ                 | 142        |
| 14.3.1    | 組み込み AMD A29K             | 142        |
| 14.3.1.1  | A29K UDI                  | 142        |
| 14.3.2    | AMD29K の EBMON プロトコル      | 142        |
| 14.3.2.1  | 通信セットアップ                  | 143        |
| 14.3.2.2  | EB29K クロス・デバッグ            | 144        |
| 14.3.2.3  | リモート・ログ                   | 144        |
| 14.3.3    | ARM                       | 145        |
| 14.3.4    | 日立 H8/300                 | 145        |
| 14.3.4.1  | 日立ボードへの接続                 | 145        |
| 14.3.4.2  | E7000 インサーキット・エミュレータの使用   | 147        |
| 14.3.4.3  | 日立マイクロプロセッサ用の特別な GDB コマンド | 147        |
| 14.3.5    | H8/500                    | 148        |
| 14.3.6    | Intel i960                | 148        |
| 14.3.6.1  | Nindy 使用時の起動方法            | 148        |
| 14.3.6.2  | Nindy 用のオプション             | 148        |
| 14.3.6.3  | Nindy reset コマンド          | 149        |
| 14.3.7    | 三菱 M32R/D                 | 149        |
| 14.3.8    | M68k                      | 149        |
| 14.3.9    | M88K                      | 150        |
| 14.3.10   | 組み込み MIPS                 | 150        |
| 14.3.11   | PowerPC                   | 152        |
| 14.3.12   | 組み込み HP PA                | 152        |
| 14.3.13   | 日立 SH                     | 152        |
| 14.3.14   | Tsquare Sparclet          | 153        |
| 14.3.14.1 | デバッグするファイルの選択             | 153        |
| 14.3.14.2 | Sparclet への接続             | 153        |
| 14.3.14.3 | Sparclet ダウンロード           | 154        |
| 14.3.14.4 | 実行とデバッグ                   | 154        |
| 14.3.15   | 富士通 Sparclite             | 154        |
| 14.3.16   | Tandem ST2000             | 154        |
| 14.3.17   | Zilog Z8000               | 155        |
| 14.4      | アーキテクチャ                   | 156        |
| 14.4.1    | A29K                      | 156        |
| 14.4.2    | Alpha                     | 156        |
| 14.4.3    | MIPS                      | 156        |

|           |                               |            |
|-----------|-------------------------------|------------|
| <b>15</b> | <b>GDB の制御</b>                | <b>157</b> |
| 15.1      | プロンプト                         | 157        |
| 15.2      | コマンド編集                        | 157        |
| 15.3      | コマンド履歴                        | 157        |
| 15.4      | 画面サイズ                         | 159        |
| 15.5      | 数値                            | 159        |
| 15.6      | オプションの警告およびメッセージ              | 160        |
| 15.7      | 内部的な事象に関するオプションのメッセージ         | 161        |
| <b>16</b> | <b>一連のコマンドのグループ化</b>          | <b>163</b> |
| 16.1      | ユーザ定義コマンド                     | 163        |
| 16.2      | ユーザ定義コマンド・フック                 | 164        |
| 16.3      | コマンド・ファイル                     | 165        |
| 16.4      | 制御された出力を得るためのコマンド             | 165        |
| <b>17</b> | <b>GNU Emacs 中での GDB の使い方</b> | <b>167</b> |
| <b>18</b> | <b>GDB 註釈</b>                 | <b>171</b> |
| 18.1      | 註釈                            | 171        |
| 18.2      | server 接頭語                    | 171        |
| 18.3      | 値                             | 172        |
| 18.4      | フレーム                          | 173        |
| 18.5      | 表示                            | 175        |
| 18.6      | GDB の入力に対する註釈                 | 175        |
| 18.7      | エラー                           | 176        |
| 18.8      | ブレイクポイントに関する情報                | 176        |
| 18.9      | 失効通知                          | 177        |
| 18.10     | プログラムの実行                      | 177        |
| 18.11     | ソースの表示                        | 178        |
| 18.12     | 将来追加する可能性のある註釈                | 178        |
| <b>19</b> | <b>GDB/MI インターフェイス</b>        | <b>179</b> |
|           | 機能と目的                         | 179        |
|           | 表記法と用語                        | 179        |
| 19.1      | GDB/MI コマンド構文                 | 179        |
| 19.1.1    | GDB/MI 入力構文                   | 179        |
| 19.1.2    | GDB/MI 出力構文                   | 180        |
| 19.1.3    | GDB/MI とのやりとりの簡単な例            | 182        |
| 19.2      | CLI に対する GDB/MI の互換性          | 183        |
| 19.3      | GDB/MI 出力レコード                 | 183        |
| 19.3.1    | GDB/MI 結果レコード                 | 183        |
| 19.3.2    | GDB/MI ストリーム・レコード             | 183        |
| 19.3.3    | GDB/MI 帯域外レコード                | 184        |
| 19.4      | GDB/MI コマンド記述フォーマット           | 184        |
| 19.5      | GDB/MI ブレイクポイント・テーブル・コマンド     | 185        |
| 19.6      | GDB/MI データ操作                  | 192        |
| 19.7      | GDB/MI プログラム制御                | 202        |

|             |                                 |            |
|-------------|---------------------------------|------------|
| 19.8        | GDB/MI におけるその他の GDB コマンド .....  | 213        |
| 19.9        | GDB/MI におけるスタック操作コマンド .....     | 214        |
| 19.10       | GDB/MI のシンボル・クエリー・コマンド .....    | 219        |
| 19.11       | GDB/MI のターゲット操作コマンド .....       | 223        |
| 19.12       | GDB/MI スレッド・コマンド .....          | 227        |
| 19.13       | GDB/MI トレースポイント・コマンド .....      | 229        |
| 19.14       | GDB/MI 変数オブジェクト .....           | 229        |
| 19.15       | GDB/MI 出力構文の変更案 .....           | 233        |
| <b>20</b>   | <b>GDB のバグ報告 .....</b>          | <b>237</b> |
| 20.1        | 本当にバグを見つけたのかどうかを知る方法 .....      | 237        |
| 20.2        | バグの報告方法 .....                   | 237        |
| <b>21</b>   | <b>コマンドライン編集 .....</b>          | <b>241</b> |
| 21.1        | 行編集入門 .....                     | 241        |
| 21.2        | Readline の操作 .....              | 241        |
| 21.2.1      | Readline の基本 .....              | 241        |
| 21.2.2      | Readline 移動コマンド .....           | 242        |
| 21.2.3      | Readline キル ( kill ) コマンド ..... | 242        |
| 21.2.4      | Readline の引数 .....              | 243        |
| 21.2.5      | 履歴中のコマンドの検索 .....               | 243        |
| 21.3        | Readline 初期化ファイル .....          | 244        |
| 21.3.1      | Readline 初期化ファイルの構文 .....       | 244        |
| 21.3.2      | 条件初期化構文 .....                   | 248        |
| 21.3.3      | 初期化ファイルのサンプル .....              | 249        |
| 21.4        | バインド可能な Readline コマンド .....     | 252        |
| 21.4.1      | 移動のためのコマンド .....                | 252        |
| 21.4.2      | 履歴を操作するためのコマンド .....            | 252        |
| 21.4.3      | テキストを変更するためのコマンド .....          | 253        |
| 21.4.4      | キルとヤンク .....                    | 254        |
| 21.4.5      | 数値引数の指定 .....                   | 256        |
| 21.4.6      | Readline による入力補完 .....          | 256        |
| 21.4.7      | キーボード・マクロ .....                 | 257        |
| 21.4.8      | その他のコマンド .....                  | 257        |
| 21.5        | Readline の vi モード .....         | 258        |
| <b>22</b>   | <b>対話的な履歴の使い方 .....</b>         | <b>259</b> |
| 22.1        | 履歴展開 .....                      | 259        |
| 22.1.1      | イベント指定子 .....                   | 259        |
| 22.1.2      | ワード指定子 .....                    | 259        |
| 22.1.3      | 修飾子 .....                       | 260        |
| <b>付録 A</b> | <b>ドキュメントのフォーマット .....</b>      | <b>261</b> |

|              |                            |     |
|--------------|----------------------------|-----|
| 付録 B         | GDB のインストール .....          | 263 |
| B.1          | 別ディレクトリでの GDB のコンパイル ..... | 264 |
| B.2          | ホストとターゲットの名前の指定 .....      | 265 |
| B.3          | configure オプション .....      | 266 |
| インデックス ..... |                            | 269 |

