Debugging with GDB

The GNU Source-Level Debugger

Tenth Edition, for GDB version AlmaLinux 12.1-5.el9

(GDB)

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

(Send bugs and comments on GDB to `https://www.gnu.org/software/gdb/bugs/`.)
Debugging with GDB
TEXinfo 2019-09-20.22

Published by the Free Software Foundation
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA
ISBN 978-0-9831592-3-0

Copyright © 1988-2022 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "Free Software" and "Free Software Needs Free Documentation", with the Front-Cover Texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below.

(a) The FSF's Back-Cover Text is: "You are free to copy and modify this GNU Manual. Buying copies from GNU Press supports the FSF in developing GNU and promoting software freedom."

Table of Contents

Summary of GDB 1
Free Software 1
Free Software Needs Free Documentation 1
Contributors to GDB 3

1 A Sample GDB Session 7

2 Getting In and Out of GDB 11
2.1 Invoking GDB 11
2.1.1 Choosing Files 12
2.1.2 Choosing Modes 13
2.1.3 What GDB Does During Startup 16
2.1.4 Initialization Files 17
2.1.4.1 Home directory early initialization files 17
2.1.4.2 System wide initialization files 18
2.1.4.3 Home directory initialization file 18
2.1.4.4 Local directory initialization file 19
2.2 Quitting GDB 19
2.3 Shell Commands 19
2.4 Logging Output 20

3 GDB Commands 23
3.1 Command Syntax 23
3.2 Command Settings 23
3.3 Command Completion 24
3.4 Command options 27
3.5 Getting Help 28

4 Running Programs Under GDB 31
4.1 Compiling for Debugging 31
4.2 Starting your Program 32
4.3 Your Program's Arguments 36
4.4 Your Program's Environment 36
4.5 Your Program's Working Directory 37
4.6 Your Program's Input and Output 38
4.7 Debugging an Already-running Process 39
4.8 Killing the Child Process 40
4.9 Debugging Multiple Inferiors Connections and Programs 40
4.10 Debugging Programs with Multiple Threads 44
4.11 Debugging Forks 49
4.12 Setting a *Bookmark* to Return to Later 52
4.12.1 A Non-obvious Benefit of Using Checkpoints 53

5 Stopping and Continuing 55
5.1 Breakpoints, Watchpoints, and Catchpoints 55
5.1.1 Setting Breakpoints 56
5.1.2 Setting Watchpoints 63
5.1.3 Setting Catchpoints 66
5.1.4 Deleting Breakpoints 70
5.1.5 Disabling Breakpoints 71
5.1.6 Break Conditions 72
5.1.7 Breakpoint Command Lists 74
5.1.8 Dynamic Printf 75
5.1.9 How to save breakpoints to a file 77
5.1.10 Static Probe Points 77
5.1.11 "Cannot insert breakpoints" 79
5.1.12 "Breakpoint address adjusted..." 79
5.2 Continuing and Stepping 80
5.3 Skipping Over Functions and Files 83
5.4 Signals 86
5.5 Stopping and Starting Multi-thread Programs 89
5.5.1 All-Stop Mode 90
5.5.2 Non-Stop Mode 91
5.5.3 Background Execution 92
5.5.4 Thread-Specific Breakpoints 93
5.5.5 Interrupted System Calls 93
5.5.6 Observer Mode 94

6 Running programs backward 97

7 Recording Inferior's Execution and Replaying It 99

8 Examining the Stack 107
8.1 Stack Frames 107
8.2 Backtraces 108
8.3 Selecting a Frame 111
8.4 Information About a Frame 113
8.5 Applying a Command to Several Frames. 114
8.6 Management of Frame Filters. 116

9 Examining Source Files 119
9.1 Printing Source Lines 119
9.2 Specifying a Location 120
9.2.1 Linespec Locations 120
9.2.2 Explicit Locations 121
9.2.3 Address Locations 122
9.3 Editing Source Files 123
9.3.1 Choosing your Editor 123

9.4 Searching Source Files ... 124
9.5 Specifying Source Directories ... 124
9.6 Source and Machine Code ... 128
9.7 Disable Reading Source Code ... 132

10 Examining Data ... 133

10.1 Expressions ... 137
10.2 Ambiguous Expressions ... 137
10.3 Program Variables ... 138
10.4 Artificial Arrays ... 141
10.5 Output Formats ... 142
10.6 Examining Memory ... 143
10.7 Memory Tagging ... 146
10.8 Automatic Display ... 147
10.9 Print Settings ... 149
10.10 Pretty Printing ... 159
10.10.1 Pretty-Printer Introduction ... 159
10.10.2 Pretty-Printer Example ... 159
10.10.3 Pretty-Printer Commands ... 160
10.11 Value History ... 161
10.12 Convenience Variables ... 162
10.13 Convenience Functions ... 165
10.14 Registers ... 169
10.15 Floating Point Hardware ... 171
10.16 Vector Unit ... 171
10.17 Operating System Auxiliary Information ... 171
10.18 Memory Region Attributes ... 173
10.18.1 Attributes ... 174
10.18.1.1 Memory Access Mode ... 174
10.18.1.2 Memory Access Size ... 174
10.18.1.3 Data Cache ... 175
10.18.2 Memory Access Checking ... 175
10.19 Copy Between Memory and a File ... 175
10.20 How to Produce a Core File from Your Program ... 176
10.21 Character Sets ... 177
10.22 Caching Data of Targets ... 180
10.23 Search Memory ... 181
10.24 Value Sizes ... 182

11 Debugging Optimized Code ... 185

11.1 Inline Functions ... 185
11.2 Tail Call Frames ... 186

12 C Preprocessor Macros ... 189

13 Tracepoints ... 193
13.1 Commands to Set Tracepoints ... 193
13.1.1 Create and Delete Tracepoints ... 194
13.1.2 Enable and Disable Tracepoints ... 196
13.1.3 Tracepoint Passcounts ... 196
13.1.4 Tracepoint Conditions ... 197
13.1.5 Trace State Variables ... 197
13.1.6 Tracepoint Action Lists ... 198
13.1.7 Listing Tracepoints ... 200
13.1.8 Listing Static Tracepoint Markers ... 201
13.1.9 Starting and Stopping Trace Experiments ... 202
13.1.10 Tracepoint Restrictions ... 204
13.2 Using the Collected Data ... 205
13.2.1 `tfind n` ... 205
13.2.2 `tdump` ... 207
13.2.3 `save tracepoints filename` ... 208
13.3 Convenience Variables for Tracepoints ... 208
13.4 Using Trace Files ... 209

14 Debugging Programs That Use Overlays ... 211
14.1 How Overlays Work ... 211
14.2 Overlay Commands ... 212
14.3 Automatic Overlay Debugging ... 214
14.4 Overlay Sample Program ... 215

15 Using GDB with Different Languages ... 217
15.1 Switching Between Source Languages ... 217
15.1.1 List of Filename Extensions and Languages ... 217
15.1.2 Setting the Working Language ... 218
15.1.3 Having GDB Infer the Source Language ... 218
15.2 Displaying the Language ... 218
15.3 Type and Range Checking ... 219
15.3.1 An Overview of Type Checking ... 219
15.3.2 An Overview of Range Checking ... 220
15.4 Supported Languages ... 221
15.4.1 C and C++ ... 221
15.4.1.1 C and C++ Operators ... 221
15.4.1.2 C and C++ Constants ... 223
15.4.1.3 C++ Expressions ... 224
15.4.1.4 C and C++ Defaults ... 225
15.4.1.5 C and C++ Type and Range Checks ... 225
15.4.1.6 GDB and C ... 225
15.4.1.7 GDB Features for C++ ... 225
15.4.1.8 Decimal Floating Point format ... 227
15.4.2 D ... 228
15.4.3 Go ... 228
15.4.4 Objective-C ... 228

15.4.4.1 Method Names in Commands 228
15.4.4.2 The Print Command With Objective-C 229
15.4.5 OpenCL C 229
15.4.5.1 OpenCL C Datatypes 229
15.4.5.2 OpenCL C Expressions 230
15.4.5.3 OpenCL C Operators 230
15.4.6 Fortran 230
15.4.6.1 Fortran Operators and Expressions 230
15.4.6.2 Fortran Defaults 230
15.4.6.3 Special Fortran Commands 230
15.4.7 Pascal 231
15.4.8 Rust 231
15.4.9 Modula-2 232
15.4.9.1 Operators 232
15.4.9.2 Built-in Functions and Procedures 233
15.4.9.3 Constants 235
15.4.9.4 Modula-2 Types 235
15.4.9.5 Modula-2 Defaults 237
15.4.9.6 Deviations from Standard Modula-2 237
15.4.9.7 Modula-2 Type and Range Checks 237
15.4.9.8 The Scope Operators :: and 238
15.4.9.9 GDB and Modula-2 238
15.4.10 Ada 238
15.4.10.1 Introduction 238
15.4.10.2 Omissions from Ada 239
15.4.10.3 Additions to Ada 240
15.4.10.4 Overloading support for Ada 242
15.4.10.5 Stopping at the Very Beginning 242
15.4.10.6 Ada Exceptions 242
15.4.10.7 Extensions for Ada Tasks 243
15.4.10.8 Tasking Support when Debugging Core Files 246
15.4.10.9 Tasking Support when using the Ravenscar Profile .. 247
15.4.10.10 Ada Source Character Set 247
15.4.10.11 Known Peculiarities of Ada Mode 248
15.5 Unsupported Languages 249

16 Examining the Symbol Table 251

17 Altering Execution 265

17.1 Assignment to Variables 265
17.2 Continuing at a Different Address 266
17.3 Giving your Program a Signal 267
17.4 Returning from a Function 268
17.5 Calling Program Functions 269
17.5.1 Calling functions with no debug info 270
17.6 Patching Programs 271
17.7 Compiling and injecting code in GDB 271

17.7.1 Compilation options for the `compile` command 273
17.7.2 Caveats when using the `compile` command 273
17.7.3 Compiler search for the `compile` command 276

18 GDB Files 277
18.1 Commands to Specify Files 277
18.2 File Caching 286
18.3 Debugging Information in Separate Files 286
18.4 Debugging information in a special section 290
18.5 Index Files Speed Up GDB 291
18.5.1 Automatic symbol index cache 292
18.6 Errors Reading Symbol Files 292
18.7 GDB Data Files 294

19 Specifying a Debugging Target 295
19.1 Active Targets 295
19.2 Commands for Managing Targets 295
19.3 Choosing Target Byte Order 298

20 Debugging Remote Programs 299
20.1 Connecting to a Remote Target 299
20.1.1 Types of Remote Connections 299
20.1.2 Host and Target Files 300
20.1.3 Remote Connection Commands 301
20.2 Sending files to a remote system 304
20.3 Using the `gdbserver` Program 304
20.3.1 Running `gdbserver` 305
20.3.1.1 Attaching to a Running Program 305
20.3.1.2 TCP port allocation lifecycle of `gdbserver` 306
20.3.1.3 Other Command-Line Arguments for `gdbserver` ... 306
20.3.2 Connecting to `gdbserver` 307
20.3.3 Monitor Commands for `gdbserver` 307
20.3.4 Tracepoints support in `gdbserver` 308
20.4 Remote Configuration 309
20.5 Implementing a Remote Stub 315
20.5.1 What the Stub Can Do for You 317
20.5.2 What You Must Do for the Stub 317
20.5.3 Putting it All Together 318

21 Configuration-Specific Information 321
21.1 Native 321
21.1.1 BSD libkvm Interface 321
21.1.2 Process Information 321
21.1.3 Features for Debugging DJGPP Programs 323
21.1.4 Features for Debugging MS Windows PE Executables ... 326
21.1.4.1 Support for DLLs without Debugging Symbols 327

21.1.4.2 DLL Name Prefixes . . . 328
21.1.4.3 Working with Minimal Symbols . . . 328
21.1.5 Commands Specific to GNU Hurd Systems . . . 329
21.1.6 Darwin . . . 331
21.1.7 FreeBSD . . . 332
21.2 Embedded Operating Systems . . . 332
21.3 Embedded Processors . . . 332
21.3.1 Synopsys ARC . . . 332
21.3.2 ARM . . . 333
21.3.3 BPF . . . 334
21.3.4 M68k . . . 334
21.3.5 MicroBlaze . . . 334
21.3.6 MIPS Embedded . . . 335
21.3.7 OpenRISC 1000 . . . 335
21.3.8 PowerPC Embedded . . . 336
21.3.9 Atmel AVR . . . 337
21.3.10 CRIS . . . 337
21.3.11 Renesas Super-H . . . 337
21.4 Architectures . . . 338
21.4.1 AArch64 . . . 338
21.4.1.1 AArch64 SVE . . . 338
21.4.1.2 AArch64 Pointer Authentication . . . 338
21.4.1.3 AArch64 Memory Tagging Extension . . . 338
21.4.2 x86 Architecture-specific Issues . . . 339
21.4.2.1 Intel *Memory Protection Extensions* (MPX) . . . 339
21.4.3 Alpha . . . 340
21.4.4 MIPS . . . 340
21.4.5 HPPA . . . 342
21.4.6 PowerPC . . . 342
21.4.7 Nios II . . . 342
21.4.8 Sparc64 . . . 343
21.4.8.1 ADI Support . . . 343
21.4.9 S12Z . . . 344

22 Controlling GDB . . . 345
22.1 Prompt . . . 345
22.2 Command Editing . . . 345
22.3 Command History . . . 346
22.4 Screen Size . . . 348
22.5 Output Styling . . . 349
22.6 Numbers . . . 351
22.7 Configuring the Current ABI . . . 352
22.8 Automatically loading associated files . . . 353
22.8.1 Automatically loading init file in the current directory . . . 354
22.8.2 Automatically loading thread debugging library . . . 355
22.8.3 Security restriction for auto-loading . . . 355
22.8.4 Displaying files tried for auto-load . . . 357
22.9 Optional Warnings and Messages . . . 357

22.10 Optional Messages about Internal Happenings 359
22.11 Other Miscellaneous Settings 364

23 Extending GDB 367
23.1 Canned Sequences of Commands 367
23.1.1 User-defined Commands 367
23.1.2 User-defined Command Hooks 370
23.1.3 Command Files 371
23.1.4 Commands for Controlled Output 372
23.1.5 Controlling auto-loading native GDB scripts 374
23.2 Command Aliases 374
23.2.1 Default Arguments 376
23.3 Extending GDB using Python 377
23.3.1 Python Commands 377
23.3.2 Python API 379
23.3.2.1 Basic Python 379
23.3.2.2 Exception Handling 384
23.3.2.3 Values From Inferior 385
23.3.2.4 Types In Python 391
23.3.2.5 Pretty Printing API 397
23.3.2.6 Selecting Pretty-Printers 398
23.3.2.7 Writing a Pretty-Printer 399
23.3.2.8 Type Printing API 401
23.3.2.9 Filtering Frames 402
23.3.2.10 Decorating Frames 404
23.3.2.11 Writing a Frame Filter 407
23.3.2.12 Unwinding Frames in Python 411
23.3.2.13 Xmethods In Python 414
23.3.2.14 Xmethod API 414
23.3.2.15 Writing an Xmethod 416
23.3.2.16 Inferiors In Python 419
23.3.2.17 Events In Python 420
23.3.2.18 Threads In Python 424
23.3.2.19 Recordings In Python 426
23.3.2.20 CLI Commands In Python 429
23.3.2.21 GDB/MI Commands In Python 433
23.3.2.22 Parameters In Python 435
23.3.2.23 Writing new convenience functions 438
23.3.2.24 Program Spaces In Python 439
23.3.2.25 Objfiles In Python 441
23.3.2.26 Accessing inferior stack frames from Python 443
23.3.2.27 Accessing blocks from Python 447
23.3.2.28 Python representation of Symbols 448
23.3.2.29 Symbol table representation in Python 452
23.3.2.30 Manipulating line tables using Python 454
23.3.2.31 Manipulating breakpoints using Python 455
23.3.2.32 Finish Breakpoints 458
23.3.2.33 Python representation of lazy strings 459

23.3.2.34 Python representation of architectures 460
23.3.2.35 Registers In Python 461
23.3.2.36 Connections In Python 462
23.3.2.37 Implementing new TUI windows 463
23.3.3 Python Auto-loading 465
23.3.4 Python modules 466
23.3.4.1 gdb.printing 466
23.3.4.2 gdb.types 466
23.3.4.3 gdb.prompt 468
23.4 Extending GDB using Guile 468
23.4.1 Guile Introduction 468
23.4.2 Guile Commands 469
23.4.3 Guile API 469
23.4.3.1 Basic Guile 470
23.4.3.2 Guile Configuration 471
23.4.3.3 GDB Scheme Data Types 472
23.4.3.4 Guile Exception Handling 473
23.4.3.5 Values From Inferior In Guile 475
23.4.3.6 Arithmetic In Guile 481
23.4.3.7 Types In Guile 482
23.4.3.8 Guile Pretty Printing API 486
23.4.3.9 Selecting Guile Pretty-Printers 488
23.4.3.10 Writing a Guile Pretty-Printer 489
23.4.3.11 Commands In Guile 491
23.4.3.12 Parameters In Guile 495
23.4.3.13 Program Spaces In Guile 497
23.4.3.14 Objfiles In Guile 498
23.4.3.15 Accessing inferior stack frames from Guile. 499
23.4.3.16 Accessing blocks from Guile. 502
23.4.3.17 Guile representation of Symbols. 503
23.4.3.18 Symbol table representation in Guile. 507
23.4.3.19 Manipulating breakpoints using Guile 508
23.4.3.20 Guile representation of lazy strings. 512
23.4.3.21 Guile representation of architectures 513
23.4.3.22 Disassembly In Guile 515
23.4.3.23 I/O Ports in Guile 515
23.4.3.24 Memory Ports in Guile 516
23.4.3.25 Iterators In Guile 517
23.4.4 Guile Auto-loading 519
23.4.5 Guile Modules 519
23.4.5.1 Guile Printing Module 519
23.4.5.2 Guile Types Module 520
23.5 Auto-loading extensions 520
23.5.1 The *objfile*-gdb.*ext* file 521
23.5.2 The .debug_gdb_scripts section 522
23.5.2.1 Script File Entries 522
23.5.2.2 Script Text Entries 523
23.5.3 Which flavor to choose? 523

23.6 Multiple Extension Languages 524
23.6.1 Python comes first 524

24 Command Interpreters 525

25 GDB Text User Interface 527
25.1 TUI Overview 527
25.2 TUI Key Bindings 528
25.3 TUI Single Key Mode 529
25.4 TUI Mouse Support 530
25.5 TUI-specific Commands 530
25.6 TUI Configuration Variables 532

26 Using GDB under GNU Emacs 535

27 The GDB/MI Interface 537
Function and Purpose 537
Notation and Terminology 537
27.3 GDB/MI General Design 537
27.3.1 Context management 538
27.3.1.1 Threads and Frames 538
27.3.1.2 Language 539
27.3.2 Asynchronous command execution and non-stop mode 539
27.3.3 Thread groups 540
27.4 GDB/MI Command Syntax 540
27.4.1 GDB/MI Input Syntax 540
27.4.2 GDB/MI Output Syntax 541
27.5 GDB/MI Compatibility with CLI 543
27.6 GDB/MI Development and Front Ends 543
27.7 GDB/MI Output Records 544
27.7.1 GDB/MI Result Records 544
27.7.2 GDB/MI Stream Records 545
27.7.3 GDB/MI Async Records 545
27.7.4 GDB/MI Breakpoint Information 549
27.7.5 GDB/MI Frame Information 552
27.7.6 GDB/MI Thread Information 552
27.7.7 GDB/MI Ada Exception Information 553
27.8 Simple Examples of GDB/MI Interaction 553
27.9 GDB/MI Command Description Format 554
27.10 GDB/MI Breakpoint Commands 554
27.11 GDB/MI Catchpoint Commands 565
27.11.1 Shared Library GDB/MI Catchpoints 565
27.11.2 Ada Exception GDB/MI Catchpoints 565
27.11.3 C++ Exception GDB/MI Catchpoints 567
27.12 GDB/MI Program Context 569
27.13 GDB/MI Thread Commands 571

27.14 GDB/MI Ada Tasking Commands 573
27.15 GDB/MI Program Execution 574
27.16 GDB/MI Stack Manipulation Commands 581
27.17 GDB/MI Variable Objects 587
27.18 GDB/MI Data Manipulation 597
27.19 GDB/MI Tracepoint Commands 606
27.20 GDB/MI Symbol Query Commands 612
27.21 GDB/MI File Commands 620
27.22 GDB/MI Target Manipulation Commands 624
27.23 GDB/MI File Transfer Commands 628
27.24 Ada Exceptions GDB/MI Commands 629
27.25 GDB/MI Support Commands 630
27.26 Miscellaneous GDB/MI Commands 632

28 GDB Annotations 641
28.1 What is an Annotation? 641
28.2 The Server Prefix 642
28.3 Annotation for GDB Input 642
28.4 Errors 643
28.5 Invalidation Notices 643
28.6 Running the Program 643
28.7 Displaying Source 644

29 JIT Compilation Interface 645
29.1 JIT Declarations 645
29.2 Registering Code 646
29.3 Unregistering Code 646
29.4 Custom Debug Info 646
29.4.1 Using JIT Debug Info Readers 647
29.4.2 Writing JIT Debug Info Readers 647

30 In-Process Agent 649
30.1 In-Process Agent Protocol 649
30.1.1 IPA Protocol Objects 650
30.1.2 IPA Protocol Commands 651

31 Reporting Bugs in GDB 653
31.1 Have You Found a Bug? 653
31.2 How to Report Bugs 653

Appendix A In Memoriam 657

Appendix B Formatting Documentation 659

Appendix C Installing GDB 661
C.1 Requirements for Building GDB 661
C.2 Invoking the GDB `configure` Script 663
C.3 Compiling GDB in Another Directory 664
C.4 Specifying Names for Hosts and Targets 665
C.5 `configure` Options 665
C.6 System-wide configuration and settings 669
C.6.1 Installed System-wide Configuration Scripts 669

Appendix D Maintenance Commands 671

Appendix E GDB Remote Serial Protocol 683
E.1 Overview 683
E.2 Packets 684
E.3 Stop Reply Packets 695
E.4 General Query Packets 699
E.5 Architecture-Specific Protocol Details 724
E.5.1 ARM-specific Protocol Details 724
E.5.1.1 ARM Breakpoint Kinds 724
E.5.1.2 ARM Memory Tag Types 725
E.5.2 MIPS-specific Protocol Details 725
E.5.2.1 MIPS Register Packet Format 725
E.5.2.2 MIPS Breakpoint Kinds 725
E.6 Tracepoint Packets 725
E.6.1 Relocate instruction reply packet 732
E.7 Host I/O Packets 732
E.8 Interrupts 734
E.9 Notification Packets 735
E.10 Remote Protocol Support for Non-Stop Mode 737
E.11 Packet Acknowledgment 737
E.12 Examples 738
E.13 File-I/O Remote Protocol Extension 738
E.13.1 File-I/O Overview 738
E.13.2 Protocol Basics 739
E.13.3 The `F` Request Packet 740
E.13.4 The `F` Reply Packet 740
E.13.5 The '`Ctrl-C`' Message 740
E.13.6 Console I/O 741
E.13.7 List of Supported Calls 741
open 741
close 743
read 743
write 743
lseek 744
rename 744
unlink 745
stat/fstat 745

gettimeofday 746
isatty 746
system 746
E.13.8 Protocol-specific Representation of Datatypes 747
Integral Datatypes 747
Pointer Values 747
Memory Transfer 747
struct stat 748
struct timeval 748
E.13.9 Constants 749
Open Flags 749
mode_t Values 749
Errno Values 749
Lseek Flags 750
Limits 750
E.13.10 File-I/O Examples 750
E.14 Library List Format 750
E.15 Library List Format for SVR4 Targets 751
E.16 Memory Map Format 752
E.17 Thread List Format 753
E.18 Traceframe Info Format 753
E.19 Branch Trace Format 754
E.20 Branch Trace Configuration Format 755

Appendix F The GDB Agent Expression Mechanism 757

F.1 General Bytecode Design 757
F.2 Bytecode Descriptions 759
F.3 Using Agent Expressions 764
F.4 Varying Target Capabilities 765
F.5 Rationale 765

Appendix G Target Descriptions 769

G.1 Retrieving Descriptions 769
G.2 Target Description Format 769
G.2.1 Inclusion 770
G.2.2 Architecture 770
G.2.3 OS ABI 771
G.2.4 Compatible Architecture 771
G.2.5 Features 771
G.2.6 Types 771
G.2.7 Registers 772
G.3 Predefined Target Types 773
G.4 Enum Target Types 774
G.5 Standard Target Features 775
G.5.1 AArch64 Features 775
G.5.2 ARC Features 775

G.5.3 ARM Features ... 776
G.5.4 i386 Features ... 777
G.5.5 LoongArch Features ... 778
G.5.6 MicroBlaze Features ... 778
G.5.7 MIPS Features ... 778
G.5.8 M68K Features ... 778
G.5.9 NDS32 Features ... 779
G.5.10 Nios II Features ... 779
G.5.11 Openrisc 1000 Features ... 779
G.5.12 PowerPC Features ... 779
G.5.13 RISC-V Features ... 780
G.5.14 RX Features ... 781
G.5.15 S/390 and System z Features ... 781
G.5.16 Sparc Features ... 782
G.5.17 TMS320C6x Features ... 782

Appendix H Operating System Information .. 783
H.1 Process list ... 783

Appendix I Trace File Format ... 785

Appendix J `.gdb_index` section format ... 787

Appendix K Download debugging resources with Debuginfod ... 791
K.1 Debuginfod Settings ... 791

Appendix L Manual pages ... 793

Appendix M GNU GENERAL PUBLIC LICENSE ... 803

Appendix N GNU Free Documentation License .. 815

Concept Index ... 823

Command, Variable, and Function Index ... 839

Summary of GDB

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes—or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C and C++. For more information, see Section 15.4 [Supported Languages], page 221. For more information, see Section 15.4.1 [C and C++], page 221.

Support for D is partial. For information on D, see Section 15.4.2 [D], page 228.

Support for Modula-2 is partial. For information on Modula-2, see Section 15.4.9 [Modula-2], page 232.

Support for OpenCL C is partial. For information on OpenCL C, see Section 15.4.5 [OpenCL C], page 229.

Debugging Pascal programs which use sets, subranges, file variables, or nested functions does not currently work. GDB does not support entering expressions, printing values, or similar features using Pascal syntax.

GDB can be used to debug programs written in Fortran, although it may be necessary to refer to some variables with a trailing underscore.

GDB can be used to debug programs written in Objective-C, using either the Apple/NeXT or the GNU Objective-C runtime.

Free Software

GDB is *free software*, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms.

Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

Free Software Needs Free Documentation

The biggest deficiency in the free software community today is not in the software—it is the lack of good free documentation that we can include with the free software. Many of our most important programs do not come with free reference manuals and free introductory

texts. Documentation is an essential part of any software package; when an important free software package does not come with a free manual and a free tutorial, that is a major gap. We have many such gaps today.

Consider Perl, for instance. The tutorial manuals that people normally use are non-free. How did this come about? Because the authors of those manuals published them with restrictive terms—no copying, no modification, source files not available—which exclude them from the free software world.

That wasn't the first time this sort of thing happened, and it was far from the last. Many times we have heard a GNU user eagerly describe a manual that he is writing, his intended contribution to the community, only to learn that he had ruined everything by signing a publication contract to make it non-free.

Free documentation, like free software, is a matter of freedom, not price. The problem with the non-free manual is not that publishers charge a price for printed copies—that in itself is fine. (The Free Software Foundation sells printed copies of manuals, too.) The problem is the restrictions on the use of the manual. Free manuals are available in source code form, and give you permission to copy and modify. Non-free manuals do not allow this.

The criteria of freedom for a free manual are roughly the same as for free software. Redistribution (including the normal kinds of commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, both on-line and on paper.

Permission for modification of the technical content is crucial too. When people modify the software, adding or changing features, if they are conscientious they will change the manual too—so they can provide accurate and clear documentation for the modified program. A manual that leaves you no choice but to write a new manual to document a changed version of the program is not really available to our community.

Some kinds of limits on the way modification is handled are acceptable. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they were modified. Even entire sections that may not be deleted or changed are acceptable, as long as they deal with nontechnical topics (like this one). These kinds of restrictions are acceptable because they don't obstruct the community's normal use of the manual.

However, it must be possible to modify all the *technical* content of the manual, and then distribute the result in all the usual media, through all the usual channels. Otherwise, the restrictions obstruct the use of the manual, it is not free, and we need another manual to replace it.

Please spread the word about this issue. Our community continues to lose manuals to proprietary publishing. If we spread the word that free software needs free reference manuals and free tutorials, perhaps the next person who wants to contribute by writing documentation will realize, before it is too late, that only free manuals contribute to the free software community.

If you are writing documentation, please insist on publishing it under the GNU Free Documentation License or another free documentation license. Remember that this decision requires your approval—you don't have to let the publisher decide. Some commercial publishers will use a free license if you insist, but they will not propose the option; it is up

to you to raise the issue and say firmly that this is what you want. If the publisher you are dealing with refuses, please try other publishers. If you're not sure whether a proposed license is free, write to `licensing@gnu.org`.

You can encourage commercial publishers to sell more free, copylefted manuals and tutorials by buying them, and particularly by buying copies from the publishers that paid for their writing or for major improvements. Meanwhile, try to avoid buying non-free documentation at all. Check the distribution terms of a manual before you buy it, and insist that whoever seeks your business must respect your freedom. Check the history of the book, and try to reward the publishers that have paid or pay the authors to work on it.

The Free Software Foundation maintains a list of free documentation published by other publishers, at `http://www.fsf.org/doc/other-free-books.html`.

Contributors to GDB

Richard Stallman was the original author of GDB, and of many other GNU programs. Many others have contributed to its development. This section attempts to credit major contributors. One of the virtues of free software is that everyone is free to contribute to it; with regret, we cannot actually acknowledge everyone here. The file `ChangeLog` in the GDB distribution approximates a blow-by-blow account.

Changes much prior to version 2.0 are lost in the mists of time.

> *Plea:* Additions to this section are particularly welcome. If you or your friends (or enemies, to be evenhanded) have been unfairly omitted from this list, we would like to add your names!

So that they may not regard their many labors as thankless, we particularly thank those who shepherded GDB through major releases: Andrew Cagney (releases 6.3, 6.2, 6.1, 6.0, 5.3, 5.2, 5.1 and 5.0); Jim Blandy (release 4.18); Jason Molenda (release 4.17); Stan Shebs (release 4.14); Fred Fish (releases 4.16, 4.15, 4.13, 4.12, 4.11, 4.10, and 4.9); Stu Grossman and John Gilmore (releases 4.8, 4.7, 4.6, 4.5, and 4.4); John Gilmore (releases 4.3, 4.2, 4.1, 4.0, and 3.9); Jim Kingdon (releases 3.5, 3.4, and 3.3); and Randy Smith (releases 3.2, 3.1, and 3.0).

Richard Stallman, assisted at various times by Peter TerMaat, Chris Hanson, and Richard Mlynarik, handled releases through 2.8.

Michael Tiemann is the author of most of the GNU C++ support in GDB, with significant additional contributions from Per Bothner and Daniel Berlin. James Clark wrote the GNU C++ demangler. Early work on C++ was by Peter TerMaat (who also did much general update work leading to release 3.0).

GDB uses the BFD subroutine library to examine multiple object-file formats; BFD was a joint project of David V. Henkel-Wallace, Rich Pixley, Steve Chamberlain, and John Gilmore.

David Johnson wrote the original COFF support; Pace Willison did the original support for encapsulated COFF.

Brent Benson of Harris Computer Systems contributed DWARF 2 support.

Adam de Boor and Bradley Davis contributed the ISI Optimum V support. Per Bothner, Noboyuki Hikichi, and Alessandro Forin contributed MIPS support. Jean-Daniel Fekete contributed Sun 386i support. Chris Hanson improved the HP9000 support. Noboyuki

Hikichi and Tomoyuki Hasei contributed Sony/News OS 3 support. David Johnson contributed Encore Umax support. Jyrki Kuoppala contributed Altos 3068 support. Jeff Law contributed HP PA and SOM support. Keith Packard contributed NS32K support. Doug Rabson contributed Acorn Risc Machine support. Bob Rusk contributed Harris Nighthawk CX-UX support. Chris Smith contributed Convex support (and Fortran debugging). Jonathan Stone contributed Pyramid support. Michael Tiemann contributed SPARC support. Tim Tucker contributed support for the Gould NP1 and Gould Powernode. Pace Willison contributed Intel 386 support. Jay Vosburgh contributed Symmetry support. Marko Mlinar contributed OpenRISC 1000 support.

Andreas Schwab contributed M68K GNU/Linux support.

Rich Schaefer and Peter Schauer helped with support of SunOS shared libraries.

Jay Fenlason and Roland McGrath ensured that GDB and GAS agree about several machine instruction sets.

Patrick Duval, Ted Goldstein, Vikram Koka and Glenn Engel helped develop remote debugging. Intel Corporation, Wind River Systems, AMD, and ARM contributed remote debugging modules for the i960, VxWorks, A29K UDI, and RDI targets, respectively.

Brian Fox is the author of the readline libraries providing command-line editing and command history.

Andrew Beers of SUNY Buffalo wrote the language-switching code, the Modula-2 support, and contributed the Languages chapter of this manual.

Fred Fish wrote most of the support for Unix System Vr4. He also enhanced the command-completion support to cover C++ overloaded symbols.

Hitachi America (now Renesas America), Ltd. sponsored the support for H8/300, H8/500, and Super-H processors.

NEC sponsored the support for the v850, Vr4xxx, and Vr5xxx processors.

Mitsubishi (now Renesas) sponsored the support for D10V, D30V, and M32R/D processors.

Toshiba sponsored the support for the TX39 Mips processor.

Matsushita sponsored the support for the MN10200 and MN10300 processors.

Fujitsu sponsored the support for SPARClite and FR30 processors.

Kung Hsu, Jeff Law, and Rick Sladkey added support for hardware watchpoints.

Michael Snyder added support for tracepoints.

Stu Grossman wrote gdbserver.

Jim Kingdon, Peter Schauer, Ian Taylor, and Stu Grossman made nearly innumerable bug fixes and cleanups throughout GDB.

The following people at the Hewlett-Packard Company contributed support for the PA-RISC 2.0 architecture, HP-UX 10.20, 10.30, and 11.0 (narrow mode), HP's implementation of kernel threads, HP's aC++ compiler, and the Text User Interface (nee Terminal User Interface): Ben Krepp, Richard Title, John Bishop, Susan Macchia, Kathy Mann, Satish Pai, India Paul, Steve Rehrauer, and Elena Zannoni. Kim Haase provided HP-specific information in this manual.

DJ Delorie ported GDB to MS-DOS, for the DJGPP project. Robert Hoehne made significant contributions to the DJGPP port.

Cygnus Solutions has sponsored GDB maintenance and much of its development since 1991. Cygnus engineers who have worked on GDB fulltime include 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, and Elena Zannoni. In addition, 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, and David Zuhn have made contributions both large and small.

Andrew Cagney, Fernando Nasser, and Elena Zannoni, while working for Cygnus Solutions, implemented the original GDB/MI interface.

Jim Blandy added support for preprocessor macros, while working for Red Hat.

Andrew Cagney designed GDB's architecture vector. Many people including Andrew Cagney, Stephane Carrez, Randolph Chung, Nick Duffek, Richard Henderson, Mark Kettenis, Grace Sainsbury, Kei Sakamoto, Yoshinori Sato, Michael Snyder, Andreas Schwab, Jason Thorpe, Corinna Vinschen, Ulrich Weigand, and Elena Zannoni, helped with the migration of old architectures to this new framework.

Andrew Cagney completely re-designed and re-implemented GDB's unwinder framework, this consisting of a fresh new design featuring frame IDs, independent frame sniffers, and the sentinel frame. Mark Kettenis implemented the DWARF 2 unwinder, Jeff Johnston the libunwind unwinder, and Andrew Cagney the dummy, sentinel, tramp, and trad unwinders. The architecture-specific changes, each involving a complete rewrite of the architecture's frame code, were carried out by Jim Blandy, Joel Brobecker, Kevin Buettner, Andrew Cagney, Stephane Carrez, Randolph Chung, Orjan Friberg, Richard Henderson, Daniel Jacobowitz, Jeff Johnston, Mark Kettenis, Theodore A. Roth, Kei Sakamoto, Yoshinori Sato, Michael Snyder, Corinna Vinschen, and Ulrich Weigand.

Christian Zankel, Ross Morley, Bob Wilson, and Maxim Grigoriev from Tensilica, Inc. contributed support for Xtensa processors. Others who have worked on the Xtensa port of GDB in the past include Steve Tjiang, John Newlin, and Scott Foehner.

Michael Eager and staff of Xilinx, Inc., contributed support for the Xilinx MicroBlaze architecture.

Initial support for the FreeBSD/mips target and native configuration was developed by SRI International and the University of Cambridge Computer Laboratory under DARPA/AFRL contract FA8750-10-C-0237 ("CTSRD"), as part of the DARPA CRASH research programme.

Initial support for the FreeBSD/riscv target and native configuration was developed by SRI International and the University of Cambridge Computer Laboratory (Department of Computer Science and Technology) under DARPA contract HR0011-18-C-0016 ("ECATS"), as part of the DARPA SSITH research programme.

The original port to the OpenRISC 1000 is believed to be due to Alessandro Forin and Per Bothner. More recent ports have been the work of Jeremy Bennett, Franck Jullien, Stefan Wallentowitz and Stafford Horne.

Weimin Pan, David Faust and Jose E. Marchesi contributed support for the Linux kernel BPF virtual architecture. This work was sponsored by Oracle.

1 A Sample GDB Session

You can use this manual at your leisure to read all about GDB. However, a handful of commands are enough to get started using the debugger. This chapter illustrates those commands.

In this sample session, we emphasize user input like this: **input**, to make it easier to pick out from the surrounding output.

One of the preliminary versions of GNU `m4` (a generic macro processor) exhibits the following bug: sometimes, when we change its quote strings from the default, the commands used to capture one macro definition within another stop working. In the following short `m4` session, we define a macro `foo` which expands to `0000`; we then use the `m4` built-in `defn` to define `bar` as the same thing. However, when we change the open quote string to `<QUOTE>` and the close quote string to `<UNQUOTE>`, the same procedure fails to define a new synonym `baz`:

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

foo
0000
define(bar,defn(‘foo’))

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

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

Let us use GDB to try to see what is going on.

```
$ 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 AlmaLinux 12.1-5.el9, Copyright 1999 Free Software Foundation, Inc...
(gdb)
```

GDB reads only enough symbol data to know where to find the rest when needed; as a result, the first prompt comes up very quickly. We now tell GDB to use a narrower display width than usual, so that examples fit in this manual.

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

We need to see how the `m4` built-in `changequote` works. Having looked at the source, we know the relevant subroutine is `m4_changequote`, so we set a breakpoint there with the GDB `break` command.

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

Using the `run` command, we start `m4` running under GDB control; as long as control does not reach the `m4_changequote` subroutine, the program runs as usual:

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

foo
0000
```

To trigger the breakpoint, we call `changequote`. GDB suspends execution of `m4`, displaying information about the context where it stops.

```
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))
```

Now we use the command `n` (`next`) to advance execution to the next line of the current function.

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

`set_quotes` looks like a promising subroutine. We can go into it by using the command `s` (`step`) instead of `next`. `step` goes to the next line to be executed in *any* subroutine, so it steps into `set_quotes`.

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

The display that shows the subroutine where `m4` is now suspended (and its arguments) is called a stack frame display. It shows a summary of the stack. We can use the `backtrace` command (which can also be spelled `bt`), to see where we are in the stack as a whole: the `backtrace` command displays a stack frame for each active subroutine.

```
(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
```

We step through a few more lines to see what happens. The first two times, we can use '`s`'; the next two times we use `n` to avoid falling into the `xstrdup` subroutine.

```
(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);
```

The last line displayed looks a little odd; we can examine the variables `lquote` and `rquote` to see if they are in fact the new left and right quotes we specified. We use the command `p` (`print`) to see their values.

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

`lquote` and `rquote` are indeed the new left and right quotes. To look at some context, we can display ten lines of source surrounding the current line with the `l` (`list`) command.

```
(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
```

Let us step past the two lines that set `len_lquote` and `len_rquote`, and then examine the values of those variables.

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

That certainly looks wrong, assuming `len_lquote` and `len_rquote` are meant to be the lengths of `lquote` and `rquote` respectively. We can set them to better values using the `p` command, since it can print the value of any expression—and that expression can include subroutine calls and assignments.

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

Is that enough to fix the problem of using the new quotes with the `m4` built-in `defn`? We can allow `m4` to continue executing with the `c` (`continue`) command, and then try the example that caused trouble initially:

```
(gdb) c
Continuing.

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

baz
0000
```

Success! The new quotes now work just as well as the default ones. The problem seems to have been just the two typos defining the wrong lengths. We allow `m4` exit by giving it an EOF as input:

```
Ctrl-d
Program exited normally.
```

The message '`Program exited normally.`' is from GDB; it indicates `m4` has finished executing. We can end our GDB session with the GDB `quit` command.

```
(gdb) quit
```

2 Getting In and Out of GDB

This chapter discusses how to start GDB, and how to get out of it. The essentials are:

- type '`gdb`' to start GDB.
- type *`quit`*, *`exit`* or *`Ctrl-d`* to exit.

2.1 Invoking GDB

Invoke GDB by running the program `gdb`. Once started, GDB reads commands from the terminal until you tell it to exit.

You can also run `gdb` with a variety of arguments and options, to specify more of your debugging environment at the outset.

The command-line options described here are designed to cover a variety of situations; in some environments, some of these options may effectively be unavailable.

The most usual way to start GDB is with one argument, specifying an executable program:

```
gdb program
```

You can also start with both an executable program and a core file specified:

```
gdb program core
```

You can, instead, specify a process ID as a second argument or use option `-p`, if you want to debug a running process:

```
gdb program 1234
gdb -p 1234
```

would attach GDB to process `1234`. With option `-p` you can omit the *program* filename.

Taking advantage of the second command-line argument requires a fairly complete operating system; when you use GDB as a remote debugger attached to a bare board, there may not be any notion of "process", and there is often no way to get a core dump. GDB will warn you if it is unable to attach or to read core dumps.

You can optionally have `gdb` pass any arguments after the executable file to the inferior using `--args`. This option stops option processing.

```
gdb --args gcc -O2 -c foo.c
```

This will cause `gdb` to debug `gcc`, and to set `gcc`'s command-line arguments (see Section 4.3 [Arguments], page 36) to '`-O2 -c foo.c`'.

You can run `gdb` without printing the front material, which describes GDB's non-warranty, by specifying `--silent` (or `-q`/`--quiet`):

```
gdb --silent
```

You can further control how GDB starts up by using command-line options. GDB itself can remind you of the options available.

Type

```
gdb -help
```

to display all available options and briefly describe their use ('`gdb -h`' is a shorter equivalent).

All options and command line arguments you give are processed in sequential order. The order makes a difference when the '`-x`' option is used.

2.1.1 Choosing Files

When GDB starts, it reads any arguments other than options as specifying an executable file and core file (or process ID). This is the same as if the arguments were specified by the '`-se`' and '`-c`' (or '`-p`') options respectively. (GDB reads the first argument that does not have an associated option flag as equivalent to the '`-se`' option followed by that argument; and the second argument that does not have an associated option flag, if any, as equivalent to the '`-c`'/'`-p`' option followed by that argument.) If the second argument begins with a decimal digit, GDB will first attempt to attach to it as a process, and if that fails, attempt to open it as a corefile. If you have a corefile whose name begins with a digit, you can prevent GDB from treating it as a pid by prefixing it with `./`, e.g. `./12345`.

If GDB has not been configured to included core file support, such as for most embedded targets, then it will complain about a second argument and ignore it.

Many options have both long and short forms; both are shown in the following list. GDB also recognizes the long forms if you truncate them, so long as enough of the option is present to be unambiguous. (If you prefer, you can flag option arguments with '`--`' rather than '`-`', though we illustrate the more usual convention.)

`-symbols` *file*
`-s` *file* Read symbol table from file *file*.

`-exec` *file*
`-e` *file* Use file *file* as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

`-se` *file* Read symbol table from file *file* and use it as the executable file.

`-core` *file*
`-c` *file* Use file *file* as a core dump to examine.

`-pid` *number*
`-p` *number* Connect to process ID *number*, as with the `attach` command.

`-command` *file*
`-x` *file* Execute commands from file *file*. The contents of this file is evaluated exactly as the `source` command would. See Section 23.1.3 [Command files], page 371.

`-eval-command` *command*
`-ex` *command*
Execute a single GDB command.

This option may be used multiple times to call multiple commands. It may also be interleaved with '`-command`' as required.

```
gdb -ex 'target sim' -ex 'load' \
   -x setbreakpoints -ex 'run' a.out
```

`-init-command` *file*
`-ix` *file* Execute commands from file *file* before loading the inferior (but after loading gdbinit files). See Section 2.1.3 [Startup], page 16.

`-init-eval-command` *command*
`-iex` *command*
Execute a single GDB command before loading the inferior (but after loading gdbinit files). See Section 2.1.3 [Startup], page 16.

`-early-init-command` *file*
`-eix` *file* Execute commands from *file* very early in the initialization process, before any output is produced. See Section 2.1.3 [Startup], page 16.

`-early-init-eval-command` *command*
`-eiex` *command*
Execute a single GDB command very early in the initialization process, before any output is produced.

`-directory` *directory*
`-d` *directory*
Add *directory* to the path to search for source and script files.

`-r`
`-readnow` Read each symbol file's entire symbol table immediately, rather than the default, which is to read it incrementally as it is needed. This makes startup slower, but makes future operations faster.

`--readnever`
Do not read each symbol file's symbolic debug information. This makes startup faster but at the expense of not being able to perform symbolic debugging. DWARF unwind information is also not read, meaning backtraces may become incomplete or inaccurate. One use of this is when a user simply wants to do the following sequence: attach, dump core, detach. Loading the debugging information in this case is an unnecessary cause of delay.

2.1.2 Choosing Modes

You can run GDB in various alternative modes—for example, in batch mode or quiet mode.

`-nx`
`-n` Do not execute commands found in any initialization files (see Section 2.1.4 [Initialization Files], page 17).

`-nh` Do not execute commands found in any home directory initialization file (see Section 2.1.4 [Home directory initialization file], page 17). The system wide and current directory initialization files are still loaded.

`-quiet`
`-silent`
`-q` "Quiet". Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

This can also be enabled using `set startup-quietly on`. The default is `off`. Use `show startup-quietly` to see the current setting. Place `set startup-quietly on` into your early initialization file (see Section 2.1.4 [Initialization Files], page 17) to have future GDB sessions startup quietly.

`-batch` Run in batch mode. Exit with status `0` after processing all the command files specified with '`-x`' (and all commands from initialization files, if not inhibited with '`-n`'). Exit with nonzero status if an error occurs in executing the GDB commands in the command files. Batch mode also disables pagination, sets unlimited terminal width and height see Section 22.4 [Screen Size], page 348, and

acts as if *`set confirm off`* were in effect (see Section 22.9 [Messages/Warnings], page 357).

Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the message

```
Program exited normally.
```

(which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.

`-batch-silent`
: Run in batch mode exactly like '`-batch`', but totally silently. All GDB output to `stdout` is prevented (`stderr` is unaffected). This is much quieter than '`-silent`' and would be useless for an interactive session.

  This is particularly useful when using targets that give '`Loading section`' messages, for example.

  Note that targets that give their output via GDB, as opposed to writing directly to `stdout`, will also be made silent.

`-return-child-result`
: The return code from GDB will be the return code from the child process (the process being debugged), with the following exceptions:

  - GDB exits abnormally. E.g., due to an incorrect argument or an internal error. In this case the exit code is the same as it would have been without '`-return-child-result`'.
  - The user quits with an explicit value. E.g., '`quit 1`'.
  - The child process never runs, or is not allowed to terminate, in which case the exit code will be -1.

  This option is useful in conjunction with '`-batch`' or '`-batch-silent`', when GDB is being used as a remote program loader or simulator interface.

`-nowindows`
`-nw`
: "No windows". If GDB comes with a graphical user interface (GUI) built in, then this option tells GDB to only use the command-line interface. If no GUI is available, this option has no effect.

`-windows`
`-w`
: If GDB includes a GUI, then this option requires it to be used if possible.

`-cd` *`directory`*
: Run GDB using *directory* as its working directory, instead of the current directory.

`-data-directory` *`directory`*
`-D` *`directory`*
: Run GDB using *directory* as its data directory. The data directory is where GDB searches for its auxiliary files. See Section 18.7 [Data Files], page 294.

`-fullname`
`-f`
: GNU Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion

each time a stack frame is displayed (which includes each time your program stops). This recognizable format looks like two '`\032`' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two '`\032`' characters as a signal to display the source code for the frame.

`-annotate` *`level`*
: This option sets the *annotation level* inside GDB. Its effect is identical to using '`set annotate` *`level`*' (see Chapter 28 [Annotations], page 641). The annotation *level* controls how much information GDB prints together with its prompt, values of expressions, source lines, and other types of output. Level 0 is the normal, level 1 is for use when GDB is run as a subprocess of GNU Emacs, level 3 is the maximum annotation suitable for programs that control GDB, and level 2 has been deprecated.

  The annotation mechanism has largely been superseded by GDB/MI (see Chapter 27 [GDB/MI], page 537).

`--args`
: Change interpretation of command line so that arguments following the executable file are passed as command line arguments to the inferior. This option stops option processing.

`-baud` *`bps`*
`-b` *`bps`*
: Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

`-l` *`timeout`*
: Set the timeout (in seconds) of any communication used by GDB for remote debugging.

`-tty` *`device`*
`-t` *`device`*
: Run using *device* for your program's standard input and output.

`-tui`
: Activate the *Text User Interface* when starting. The Text User Interface manages several text windows on the terminal, showing source, assembly, registers and GDB command outputs (see Chapter 25 [GDB Text User Interface], page 527). Do not use this option if you run GDB from Emacs (see Chapter 26 [Using GDB under GNU Emacs], page 535).

`-interpreter` *`interp`*
: Use the interpreter *interp* for interface with the controlling program or device. This option is meant to be set by programs which communicate with GDB using it as a back end. See Chapter 24 [Command Interpreters], page 525.

  '`--interpreter=mi`' (or '`--interpreter=mi3`') causes GDB to use the GDB/MI *interface* version 3 (see Chapter 27 [The GDB/MI Interface], page 537) included since GDB version 9.1. GDB/MI version 2 (`mi2`), included in GDB 6.0 and version 1 (`mi1`), included in GDB 5.3, are also available. Earlier GDB/MI interfaces are no longer supported.

`-write`
: Open the executable and core files for both reading and writing. This is equivalent to the '`set write on`' command inside GDB (see Section 17.6 [Patching], page 271).

`-statistics`
: This option causes GDB to print statistics about time and memory usage after it completes each command and returns to the prompt.

`-version`
: This option causes GDB to print its version number and no-warranty blurb, and exit.

`-configuration`
: This option causes GDB to print details about its build-time configuration parameters, and then exit. These details can be important when reporting GDB bugs (see Chapter 31 [GDB Bugs], page 653).

2.1.3 What GDB Does During Startup

Here's the description of what GDB does during session startup:

1. Performs minimal setup required to initialize basic internal state.
2. Reads commands from the early initialization file (if any) in your home directory. Only a restricted set of commands can be placed into an early initialization file, see Section 2.1.4 [Initialization Files], page 17, for details.
3. Executes commands and command files specified by the '`-eiex`' and '`-eix`' command line options in their specified order. Only a restricted set of commands can be used with '`-eiex`' and '`eix`', see Section 2.1.4 [Initialization Files], page 17, for details.
4. Sets up the command interpreter as specified by the command line (see Section 2.1.2 [Mode Options], page 13).
5. Reads the system wide initialization file and the files from the system wide initialization directory, see [System Wide Init Files], page 18.
6. Reads the initialization file (if any) in your home directory and executes all the commands in that file, see [Home Directory Init File], page 18.
7. Executes commands and command files specified by the '`-iex`' and '`-ix`' options in their specified order. Usually you should use the '`-ex`' and '`-x`' options instead, but this way you can apply settings before GDB init files get executed and before inferior gets loaded.
8. Processes command line options and operands.
9. Reads and executes the commands from the initialization file (if any) in the current working directory as long as '`set auto-load local-gdbinit`' is set to '`on`' (see Section 22.8.1 [Init File in the Current Directory], page 354). This is only done if the current directory is different from your home directory. Thus, you can have more than one init file, one generic in your home directory, and another, specific to the program you are debugging, in the directory where you invoke GDB. See [Init File in the Current Directory during Startup], page 19.
10. If the command line specified a program to debug, or a process to attach to, or a core file, GDB loads any auto-loaded scripts provided for the program or for its loaded shared libraries. See Section 22.8 [Auto-loading], page 353.

    If you wish to disable the auto-loading during startup, you must do something like the following:

    ```
    $ gdb -iex "set auto-load python-scripts off" myprogram
    ```

    Option '`-ex`' does not work because the auto-loading is then turned off too late.

11. Executes commands and command files specified by the '`-ex`' and '`-x`' options in their specified order. See Section 23.1.3 [Command Files], page 371, for more details about GDB command files.
12. Reads the command history recorded in the *history file*. See Section 22.3 [Command History], page 346, for more details about the command history and the files where GDB records it.

2.1.4 Initialization Files

During startup (see Section 2.1.3 [Startup], page 16) GDB will execute commands from several initialization files. These initialization files use the same syntax as *command files* (see Section 23.1.3 [Command Files], page 371) and are processed by GDB in the same way.

To display the list of initialization files loaded by GDB at startup, in the order they will be loaded, you can use *`gdb --help`*.

The *early initialization* file is loaded very early in GDB's initialization process, before the interpreter (see Chapter 24 [Interpreters], page 525) has been initialized, and before the default target (see Chapter 19 [Targets], page 295) is initialized. Only `set` or `source` commands should be placed into an early initialization file, and the only `set` commands that can be used are those that control how GDB starts up.

Commands that can be placed into an early initialization file will be documented as such throughout this manual. Any command that is not documented as being suitable for an early initialization file should instead be placed into a general initialization file. Command files passed to `--early-init-command` or `-eix` are also early initialization files, with the same command restrictions. Only commands that can appear in an early initialization file should be passed to `--early-init-eval-command` or `-eiex`.

In contrast, the *general initialization* files are processed later, after GDB has finished its own internal initialization process, any valid command can be used in these files.

Throughout the rest of this document the term *initialization file* refers to one of the general initialization files, not the early initialization file. Any discussion of the early initialization file will specifically mention that it is the early initialization file being discussed.

As the system wide and home directory initialization files are processed before most command line options, changes to settings (e.g. '`set complaints`') can affect subsequent processing of command line options and operands.

The following sections describe where GDB looks for the early initialization and initialization files, and the order that the files are searched for.

2.1.4.1 Home directory early initialization files

GDB initially looks for an early initialization file in the users home directory[1]. There are a number of locations that GDB will search in the home directory, these locations are searched in order and GDB will load the first file that it finds, and subsequent locations will not be checked.

On non-macOS hosts the locations searched are:

- The file `gdb/gdbearlyinit` within the directory pointed to by the environment variable `XDG_CONFIG_HOME`, if it is defined.

[1] On DOS/Windows systems, the home directory is the one pointed to by the `HOME` environment variable.

- The file `.config/gdb/gdbearlyinit` within the directory pointed to by the environment variable `HOME`, if it is defined.
- The file `.gdbearlyinit` within the directory pointed to by the environment variable `HOME`, if it is defined.

By contrast, on macOS hosts the locations searched are:

- The file `Library/Preferences/gdb/gdbearlyinit` within the directory pointed to by the environment variable `HOME`, if it is defined.
- The file `.gdbearlyinit` within the directory pointed to by the environment variable `HOME`, if it is defined.

It is possible to prevent the home directory early initialization file from being loaded using the '`-nx`' or '`-nh`' command line options, see Section 2.1.2 [Choosing Modes], page 13.

2.1.4.2 System wide initialization files

There are two locations that are searched for system wide initialization files. Both of these locations are always checked:

`system.gdbinit`
: This is a single system-wide initialization file. Its location is specified with the `--with-system-gdbinit` configure option (see Section C.6 [System-wide configuration], page 669). It is loaded first when GDB starts, before command line options have been processed.

`system.gdbinit.d`
: This is the system-wide initialization directory. Its location is specified with the `--with-system-gdbinit-dir` configure option (see Section C.6 [System-wide configuration], page 669). Files in this directory are loaded in alphabetical order immediately after `system.gdbinit` (if enabled) when GDB starts, before command line options have been processed. Files need to have a recognized scripting language extension (`.py`/`.scm`) or be named with a `.gdb` extension to be interpreted as regular GDB commands. GDB will not recurse into any subdirectories of this directory.

It is possible to prevent the system wide initialization files from being loaded using the '`-nx`' command line option, see Section 2.1.2 [Choosing Modes], page 13.

2.1.4.3 Home directory initialization file

After loading the system wide initialization files GDB will look for an initialization file in the users home directory[2]. There are a number of locations that GDB will search in the home directory, these locations are searched in order and GDB will load the first file that it finds, and subsequent locations will not be checked.

On non-Apple hosts the locations searched are:

```
$XDG_CONFIG_HOME/gdb/gdbinit
$HOME/.config/gdb/gdbinit
$HOME/.gdbinit
```

While on Apple hosts the locations searched are:

[2] On DOS/Windows systems, the home directory is the one pointed to by the `HOME` environment variable.

`$HOME/Library/Preferences/gdb/gdbinit`
`$HOME/.gdbinit`

It is possible to prevent the home directory initialization file from being loaded using the '`-nx`' or '`-nh`' command line options, see Section 2.1.2 [Choosing Modes], page 13.

The DJGPP port of GDB uses the name `gdb.ini` instead of `.gdbinit` or `gdbinit`, due to the limitations of file names imposed by DOS filesystems. The Windows port of GDB uses the standard name, but if it finds a `gdb.ini` file in your home directory, it warns you about that and suggests to rename the file to the standard name.

2.1.4.4 Local directory initialization file

GDB will check the current directory for a file called `.gdbinit`. It is loaded last, after command line options other than '`-x`' and '`-ex`' have been processed. The command line options '`-x`' and '`-ex`' are processed last, after `.gdbinit` has been loaded, see Section 2.1.1 [Choosing Files], page 12.

If the file in the current directory was already loaded as the home directory initialization file then it will not be loaded a second time.

It is possible to prevent the local directory initialization file from being loaded using the '`-nx`' command line option, see Section 2.1.2 [Choosing Modes], page 13.

2.2 Quitting GDB

`quit` [*expression*]
`exit` [*expression*]
`q`

To exit GDB, use the `quit` command (abbreviated `q`), the `exit` command, or type an end-of-file character (usually *`Ctrl-d`*). If you do not supply *expression*, GDB will terminate normally; otherwise it will terminate using the result of *expression* as the error code.

An interrupt (often *`Ctrl-c`*) does not exit from GDB, but rather terminates the action of any GDB command that is in progress and returns to GDB command level. It is safe to type the interrupt character at any time because GDB does not allow it to take effect until a time when it is safe.

If you have been using GDB to control an attached process or device, you can release it with the `detach` command (see Section 4.7 [Debugging an Already-running Process], page 39).

2.3 Shell Commands

If you need to execute occasional shell commands during your debugging session, there is no need to leave or suspend GDB; you can just use the `shell` command.

`shell` *command-string*
`!`*command-string*

Invoke a standard shell to execute *command-string*. Note that no space is needed between `!` and *command-string*. On GNU and Unix systems, the environment variable `SHELL`, if it exists, determines which shell to run. Otherwise GDB uses the default shell (`/bin/sh` on GNU and Unix systems, `cmd.exe` on MS-Windows, `COMMAND.COM` on MS-DOS, etc.).

The utility `make` is often needed in development environments. You do not have to use the `shell` command for this purpose in GDB:

`make` *`make-args`*
: Execute the `make` program with the specified arguments. This is equivalent to '`shell make` *`make-args`*'.

`pipe [`*`command`*`] | `*`shell_command`*
`| [`*`command`*`] | `*`shell_command`*
`pipe -d` *`delim command delim shell_command`*
`| -d` *`delim command delim shell_command`*
: Executes *command* and sends its output to *shell_command*. Note that no space is needed around `|`. If no *command* is provided, the last command executed is repeated.

  In case the *command* contains a `|`, the option `-d` *`delim`* can be used to specify an alternate delimiter string *delim* that separates the *command* from the *shell_command*.

  Example:

```
(gdb) p var
$1 = {
  black = 144,
  red = 233,
  green = 377,
  blue = 610,
  white = 987
}
(gdb) pipe p var|wc
      7      19      80
(gdb) |p var|wc -l
7
(gdb) p /x var
$4 = {
  black = 0x90,
  red = 0xe9,
  green = 0x179,
  blue = 0x262,
  white = 0x3db
}
(gdb) ||grep red
  red => 0xe9,
(gdb) | -d ! echo this contains a | char\n ! sed -e 's/|/PIPE/'
this contains a PIPE char
(gdb) | -d xxx echo this contains a | char!\n xxx sed -e 's/|/PIPE/'
this contains a PIPE char!
(gdb)
```

The convenience variables `$_shell_exitcode` and `$_shell_exitsignal` can be used to examine the exit status of the last shell command launched by `shell`, `make`, `pipe` and `|`. See Section 10.12 [Convenience Variables], page 162.

2.4 Logging Output

You may want to save the output of GDB commands to a file. There are several commands to control GDB's logging.

`set logging enabled [on|off]`
: Enable or disable logging.

`set logging file` *file*
: Change the name of the current logfile. The default logfile is `gdb.txt`.

`set logging overwrite [on|off]`
: By default, GDB will append to the logfile. Set `overwrite` if you want `set logging enabled on` to overwrite the logfile instead.

`set logging redirect [on|off]`
: By default, GDB output will go to both the terminal and the logfile. Set `redirect` if you want output to go only to the log file.

`set logging debugredirect [on|off]`
: By default, GDB debug output will go to both the terminal and the logfile. Set `debugredirect` if you want debug output to go only to the log file.

`show logging`
: Show the current values of the logging settings.

You can also redirect the output of a GDB command to a shell command. See [pipe], page 20.

3 GDB Commands

You can abbreviate a GDB command to the first few letters of the command name, if that abbreviation is unambiguous; and you can repeat certain GDB commands by typing just `RET`. You can also use the `TAB` key to get GDB to fill out the rest of a word in a command (or to show you the alternatives available, if there is more than one possibility).

3.1 Command Syntax

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument which is the number of times to step, as in '`step 5`'. You can also use the `step` command with no arguments. Some commands do not allow any arguments.

GDB command names may always be truncated if that abbreviation is unambiguous. Other possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, `s` is specially defined as equivalent to `step` even though there are other commands whose names start with `s`. You can test abbreviations by using them as arguments to the `help` command.

A blank line as input to GDB (typing just `RET`) means to repeat the previous command. Certain commands (for example, `run`) will not repeat this way; these are commands whose unintentional repetition might cause trouble and which you are unlikely to want to repeat. User-defined commands can disable this feature; see Section 23.1.1 [Define], page 367.

The `list` and `x` commands, when you repeat them with `RET`, construct new arguments rather than repeating exactly as typed. This permits easy scanning of source or memory.

GDB can also use `RET` in another way: to partition lengthy output, in a way similar to the common utility `more` (see Section 22.4 [Screen Size], page 348). Since it is easy to press one `RET` too many in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a `#` to the end of the line is a comment; it does nothing. This is useful mainly in command files (see Section 23.1.3 [Command Files], page 371).

The *`Ctrl-o`* binding is useful for repeating a complex sequence of commands. This command accepts the current line, like `RET`, and then fetches the next line relative to the current line from the history for editing.

3.2 Command Settings

Many commands change their behavior according to command-specific variables or settings. These settings can be changed with the `set` subcommands. For example, the `print` command (see Chapter 10 [Examining Data], page 133) prints arrays differently depending on settings changeable with the commands `set print elements NUMBER-OF-ELEMENTS` and `set print array-indexes`, among others.

You can change these settings to your preference in the gdbinit files loaded at GDB startup. See Section 2.1.3 [Startup], page 16.

The settings can also be changed interactively during the debugging session. For example, to change the limit of array elements to print, you can do the following:

```
(GDB) set print elements 10
```

```
(GDB) print some_array
$1 = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90...}
```

The above `set print elements 10` command changes the number of elements to print from the default of 200 to 10. If you only intend this limit of 10 to be used for printing `some_array`, then you must restore the limit back to 200, with `set print elements 200`.

Some commands allow overriding settings with command options. For example, the `print` command supports a number of options that allow overriding relevant global print settings as set by `set print` subcommands. See [print options], page 133. The example above could be rewritten as:

```
(GDB) print -elements 10 -- some_array
$1 = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90...}
```

Alternatively, you can use the `with` command to change a setting temporarily, for the duration of a command invocation.

`with` *`setting`* [*`value`*] [`--` *`command`*]
`w` *`setting`* [*`value`*] [`--` *`command`*]

Temporarily set *setting* to *value* for the duration of *command*.

setting is any setting you can change with the `set` subcommands. *value* is the value to assign to `setting` while running `command`.

If no *command* is provided, the last command executed is repeated.

If a *command* is provided, it must be preceded by a double dash (`--`) separator. This is required because some settings accept free-form arguments, such as expressions or filenames.

For example, the command

```
(GDB) with print array on -- print some_array
```

is equivalent to the following 3 commands:

```
(GDB) set print array on
(GDB) print some_array
(GDB) set print array off
```

The `with` command is particularly useful when you want to override a setting while running user-defined commands, or commands defined in Python or Guile. See Chapter 23 [Extending GDB], page 367.

```
(GDB) with print pretty on -- my_complex_command
```

To change several settings for the same command, you can nest `with` commands. For example, `with language ada -- with print elements 10` temporarily changes the language to Ada and sets a limit of 10 elements to print for arrays and strings.

3.3 Command Completion

GDB can fill in the rest of a word in a command for you, if there is only one possibility; it can also show you what the valid possibilities are for the next word in a command, at any time. This works for GDB commands, GDB subcommands, command options, and the names of symbols in your program.

Press the `TAB` key whenever you want GDB to fill out the rest of a word. If there is only one possibility, GDB fills in the word, and waits for you to finish the command (or press `RET` to enter it). For example, if you type

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

GDB fills in the rest of the word '`breakpoints`', since that is the only `info` subcommand beginning with '`bre`':

```
(gdb) info breakpoints
```

You can either press `RET` at this point, to run the `info breakpoints` command, or backspace and enter something else, if '`breakpoints`' does not look like the command you expected. (If you were sure you wanted `info breakpoints` in the first place, you might as well just type `RET` immediately after '`info bre`', to exploit command abbreviations rather than command completion).

If there is more than one possibility for the next word when you press `TAB`, GDB sounds a bell. You can either supply more characters and try again, or just press `TAB` a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with '`make_`', but when you type *`b make_TAB`* GDB just sounds the bell. Typing `TAB` again displays all the function names in your program that begin with those characters, for example:

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

GDB sounds bell; press `TAB` again, to see:

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

After displaying the available possibilities, GDB copies your partial input ('`b make_`' in the example) so you can finish the command.

If you just want to see the list of alternatives in the first place, you can press *`M-?`* rather than pressing `TAB` twice. *`M-?`* means *`META ?`*. You can type this either by holding down a key designated as the `META` shift on your keyboard (if there is one) while typing `?`, or as `ESC` followed by `?`.

If the number of possible completions is large, GDB will print as much of the list as it has collected, as well as a message indicating that the list may be truncated.

```
(gdb) b mTABTAB
main
<... the rest of the possible completions ...>
*** List may be truncated, max-completions reached. ***
(gdb) b m
```

This behavior can be controlled with the following commands:

`set max-completions` *`limit`*
`set max-completions unlimited`

Set the maximum number of completion candidates. GDB will stop looking for more completions once it collects this many candidates. This is useful when completing on things like function names as collecting all the possible candidates can be time consuming. The default value is 200. A value of zero disables tab-completion. Note that setting either no limit or a very large limit can make completion slow.

`show max-completions`
: Show the maximum number of candidates that GDB will collect and show during completion.

Sometimes the string you need, while logically a "word", may contain parentheses or other characters that GDB normally excludes from its notion of a word. To permit word completion to work in this situation, you may enclose words in `'` (single quote marks) in GDB commands.

A likely situation where you might need this is in typing an expression that involves a C++ symbol name with template parameters. This is because when completing expressions, GDB treats the '`<`' character as word delimiter, assuming that it's the less-than comparison operator (see Section 15.4.1.1 [C and C++ Operators], page 221).

For example, when you want to call a C++ template function interactively using the `print` or `call` commands, you may need to distinguish whether you mean the version of `name` that was specialized for `int`, `name<int>()`, or the version that was specialized for `float`, `name<float>()`. To use the word-completion facilities in this situation, type a single quote `'` at the beginning of the function name. This alerts GDB that it may need to consider more information than usual when you press `TAB` or *`M-?`* to request word completion:

```
(gdb) p 'func< M-?
func<int>()    func<float>()
(gdb) p 'func<
```

When setting breakpoints however (see Section 9.2 [Specify Location], page 120), you don't usually need to type a quote before the function name, because GDB understands that you want to set a breakpoint on a function:

```
(gdb) b func< M-?
func<int>()    func<float>()
(gdb) b func<
```

This is true even in the case of typing the name of C++ overloaded functions (multiple definitions of the same function, distinguished by argument type). For example, when you want to set a breakpoint you don't need to distinguish whether you mean the version of `name` that takes an `int` parameter, `name(int)`, or the version that takes a `float` parameter, `name(float)`.

```
(gdb) b bubble( M-?
bubble(int)    bubble(double)
(gdb) b bubble(dou M-?
bubble(double)
```

See [quoting names], page 251, for a description of other scenarios that require quoting.

For more information about overloaded functions, see Section 15.4.1.3 [C++ Expressions], page 224. You can use the command `set overload-resolution off` to disable overload resolution; see Section 15.4.1.7 [GDB Features for C++], page 225.

When completing in an expression which looks up a field in a structure, GDB also tries[1] to limit completions to the field names available in the type of the left-hand-side:

```
(gdb) p gdb_stdout.M-?
magic               to_fputs            to_rewind
to_data             to_isatty           to_write
```

[1] The completer can be confused by certain kinds of invalid expressions. Also, it only examines the static type of the expression, not the dynamic type.

```
to_delete           to_put              to_write_async_safe
to_flush            to_read
```

This is because the `gdb_stdout` is a variable of the type `struct ui_file` that is defined in GDB sources as follows:

```
struct ui_file
{
   int *magic;
   ui_file_flush_ftype *to_flush;
   ui_file_write_ftype *to_write;
   ui_file_write_async_safe_ftype *to_write_async_safe;
   ui_file_fputs_ftype *to_fputs;
   ui_file_read_ftype *to_read;
   ui_file_delete_ftype *to_delete;
   ui_file_isatty_ftype *to_isatty;
   ui_file_rewind_ftype *to_rewind;
   ui_file_put_ftype *to_put;
   void *to_data;
}
```

3.4 Command options

Some commands accept options starting with a leading dash. For example, `print -pretty`. Similarly to command names, you can abbreviate a GDB option to the first few letters of the option name, if that abbreviation is unambiguous, and you can also use the `TAB` key to get GDB to fill out the rest of a word in an option (or to show you the alternatives available, if there is more than one possibility).

Some commands take raw input as argument. For example, the print command processes arbitrary expressions in any of the languages supported by GDB. With such commands, because raw input may start with a leading dash that would be confused with an option or any of its abbreviations, e.g. `print -p` (short for `print -pretty` or printing negative `p`?), if you specify any command option, then you must use a double-dash (`--`) delimiter to indicate the end of options.

Some options are described as accepting an argument which can be either `on` or `off`. These are known as *boolean options*. Similarly to boolean settings commands—`on` and `off` are the typical values, but any of `1`, `yes` and `enable` can also be used as "true" value, and any of `0`, `no` and `disable` can also be used as "false" value. You can also omit a "true" value, as it is implied by default.

For example, these are equivalent:

```
(gdb) print -object on -pretty off -element unlimited -- *myptr
(gdb) p -o -p 0 -e u -- *myptr
```

You can discover the set of options some command accepts by completing on `-` after the command name. For example:

```
(gdb) print -TABTAB
-address         -max-depth               -pretty          -symbol
-array           -memory-tag-violations   -raw-values      -union
-array-indexes   -null-stop               -repeats         -vtbl
-elements        -object                  -static-members
```

Completion will in some cases guide you with a suggestion of what kind of argument an option expects. For example:

```
(gdb) print -elements TABTAB
```

```
NUMBER     unlimited
```

Here, the option expects a number (e.g., `100`), not literal `NUMBER`. Such metasyntactical arguments are always presented in uppercase.

(For more on using the `print` command, see Chapter 10 [Examining Data], page 133.)

3.5 Getting Help

You can always ask GDB itself for information on its commands, using the command `help`.

`help`
`h`
: You can use `help` (abbreviated `h`) with no arguments to display a short list of named classes of commands:

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

aliases -- User-defined 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`*
: Using one of the general help classes as an argument, you can get a list of the individual commands in that class. If a command has aliases, the aliases are given after the command name, separated by commas. If an alias has default arguments, the full definition of the alias is given after the first line. For example, here is the help display for the class `status`:

```
(gdb) help status
Status inquiries.

List of commands:

info, inf, i -- Generic command for showing things
        about the program being debugged
info address, iamain  -- Describe where symbol SYM is stored.
  alias iamain = info address main
info all-registers -- List of all registers and their contents,
        for selected stack frame.
...
show, info set -- 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*

With a command name as `help` argument, GDB displays a short paragraph on how to use that command. If that command has one or more aliases, GDB will display a first line with the command name and all its aliases separated by commas. This first line will be followed by the full definition of all aliases having default arguments.

`apropos [-v]` *regexp*

The `apropos` command searches through all of the GDB commands, and their documentation, for the regular expression specified in *args*. It prints out all matches found. The optional flag '`-v`', which stands for '`verbose`', indicates to output the full documentation of the matching commands and highlight the parts of the documentation matching *regexp*. For example:

```
apropos alias
```

results in:

```
alias -- Define a new command that is an alias of an existing command
aliases -- User-defined aliases of other commands
```

while

```
apropos -v cut.*thread apply
```

results in the below output, where '`cut for 'thread apply`' is highlighted if styling is enabled.

```
taas -- Apply a command to all threads (ignoring errors
and empty output).
Usage: taas COMMAND
shortcut for 'thread apply all -s COMMAND'

tfaas -- Apply a command to all frames of all threads
(ignoring errors and empty output).
Usage: tfaas COMMAND
shortcut for 'thread apply all -s frame apply all -s COMMAND'
```

`complete` *args*

The `complete` *args* command lists all the possible completions for the beginning of a command. Use *args* to specify the beginning of the command you want completed. For example:

```
complete i
```

results in:

```
if
ignore
info
inspect
```

This is intended for use by GNU Emacs.

In addition to `help`, you can use the GDB commands `info` and `show` to inquire about the state of your program, or the state of GDB itself. Each command supports many topics

of inquiry; this manual introduces each of them in the appropriate context. The listings under `info` and under `show` in the Command, Variable, and Function Index point to all the sub-commands. See [Command and Variable Index], page 839.

`info` This command (abbreviated `i`) is for describing the state of your program. For example, you can show the arguments passed to a function with `info args`, list the registers currently in use with `info registers`, or list the breakpoints you have set with `info breakpoints`. You can get a complete list of the `info` sub-commands with `help info`.

`set` You can assign the result of an expression to an environment variable with `set`. For example, you can set the GDB prompt to a $-sign with `set prompt $`.

`show` In contrast to `info`, `show` is for describing the state of GDB itself. You can change most of the things you can `show`, by using the related command `set`; for example, you can control what number system is used for displays with `set radix`, or simply inquire which is currently in use with `show radix`.

To display all the settable parameters and their current values, you can use `show` with no arguments; you may also use `info set`. Both commands produce the same display.

Here are several miscellaneous `show` subcommands, all of which are exceptional in lacking corresponding `set` commands:

`show version`
Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site, you may need to determine which version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. Also, many system vendors ship variant versions of GDB, and there are variant versions of GDB in GNU/Linux distributions as well. The version number is the same as the one announced when you start GDB.

`show copying`
`info copying`
Display information about permission for copying GDB.

`show warranty`
`info warranty`
Display the GNU "NO WARRANTY" statement, or a warranty, if your version of GDB comes with one.

`show configuration`
Display detailed information about the way GDB was configured when it was built. This displays the optional arguments passed to the `configure` script and also configuration parameters detected automatically by `configure`. When reporting a GDB bug (see Chapter 31 [GDB Bugs], page 653), it is important to include this information in your report.

4 Running Programs Under GDB

When you run a program under GDB, you must first generate debugging information when you compile it.

You may start GDB with its arguments, if any, in an environment of your choice. If you are doing native debugging, you may redirect your program's input and output, debug an already running process, or kill a child process.

4.1 Compiling for Debugging

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the '`-g`' option when you run the compiler.

Programs that are to be shipped to your customers are compiled with optimizations, using the '`-O`' compiler option. However, some compilers are unable to handle the '`-g`' and '`-O`' options together. Using those compilers, you cannot generate optimized executables containing debugging information.

GCC, the GNU C/C++ compiler, supports '`-g`' with or without '`-O`', making it possible to debug optimized code. We recommend that you *always* use '`-g`' whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck. For more information, see Chapter 11 [Optimized Code], page 185.

Older versions of the GNU C compiler permitted a variant option '`-gg`' for debugging information. GDB no longer supports this format; if your GNU C compiler has this option, do not use it.

GDB knows about preprocessor macros and can show you their expansion (see Chapter 12 [Macros], page 189). Most compilers do not include information about preprocessor macros in the debugging information if you specify the `-g` flag alone. Version 3.1 and later of GCC, the GNU C compiler, provides macro information if you are using the DWARF debugging format, and specify the option `-g3`.

See Section "Options for Debugging Your Program or GCC" in *Using the* GNU *Compiler Collection (GCC)*, for more information on GCC options affecting debug information.

You will have the best debugging experience if you use the latest version of the DWARF debugging format that your compiler supports. DWARF is currently the most expressive and best supported debugging format in GDB.

4.2 Starting your Program

`run`
`r`

Use the `run` command to start your program under GDB. You must first specify the program name with an argument to GDB (see Chapter 2 [Getting In and Out of GDB], page 11), or by using the `file` or `exec-file` command (see Section 18.1 [Commands to Specify Files], page 277).

If you are running your program in an execution environment that supports processes, `run` creates an inferior process and makes that process run your program. In some environments without processes, `run` jumps to the start of your program. Other targets, like '`remote`', are always running. If you get an error message like this one:

```
The "remote" target does not support "run".
Try "help target" or "continue".
```

then use `continue` to run your program. You may need `load` first (see [load], page 297).

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do *before* starting your program. (You can change it after starting your program, but such changes only affect your program the next time you start it.) This information may be divided into four categories:

The *arguments.*

Specify the arguments to give your program as the arguments of the `run` command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the `SHELL` environment variable. If you do not define `SHELL`, GDB uses the default shell (`/bin/sh`). You can disable use of any shell with the `set startup-with-shell` command (see below for details).

The *environment.*

Your program normally inherits its environment from GDB, but you can use the GDB commands `set environment` and `unset environment` to change parts of the environment that affect your program. See Section 4.4 [Your Program's Environment], page 36.

The *working directory.*

You can set your program's working directory with the command *`set cwd`*. If you do not set any working directory with this command, your program will inherit GDB's working directory if native debugging, or the remote server's working directory if remote debugging. See Section 4.5 [Your Program's Working Directory], page 37.

The *standard input and output.*

Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the `run` command line, or you can use the `tty` command to set a different device for your program. See Section 4.6 [Your Program's Input and Output], page 38.

Warning: While input and output redirection work, you cannot use pipes to pass the output of the program you are debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

When you issue the `run` command, your program begins to execute immediately. See Chapter 5 [Stopping and Continuing], page 55, for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the `print` or `call` commands. See Chapter 10 [Examining Data], page 133.

If the modification time of your symbol file has changed since the last time GDB read its symbols, GDB discards its symbol table, and reads it again. When it does this, GDB tries to retain your current breakpoints.

`start` The name of the main procedure can vary from language to language. With C or C++, the main procedure name is always `main`, but other languages such as Ada do not require a specific name for their main procedure. The debugger provides a convenient way to start the execution of the program and to stop at the beginning of the main procedure, depending on the language used.

The '`start`' command does the equivalent of setting a temporary breakpoint at the beginning of the main procedure and then invoking the '`run`' command.

Some programs contain an *elaboration* phase where some startup code is executed before the main procedure is called. This depends on the languages used to write your program. In C++, for instance, constructors for static and global objects are executed before `main` is called. It is therefore possible that the debugger stops before reaching the main procedure. However, the temporary breakpoint will remain to halt execution.

Specify the arguments to give to your program as arguments to the '`start`' command. These arguments will be given verbatim to the underlying '`run`' command. Note that the same arguments will be reused if no argument is provided during subsequent calls to '`start`' or '`run`'.

It is sometimes necessary to debug the program during elaboration. In these cases, using the `start` command would stop the execution of your program too late, as the program would have already completed the elaboration phase. Under these circumstances, either insert breakpoints in your elaboration code before running your program or use the `starti` command.

`starti` The '`starti`' command does the equivalent of setting a temporary breakpoint at the first instruction of a program's execution and then invoking the '`run`' command. For programs containing an elaboration phase, the `starti` command will stop execution at the start of the elaboration phase.

`set exec-wrapper` *`wrapper`*
`show exec-wrapper`
`unset exec-wrapper`

When '`exec-wrapper`' is set, the specified wrapper is used to launch programs for debugging. GDB starts your program with a shell command of the form *`exec wrapper program`*. Quoting is added to *program* and its arguments, but not to *wrapper*, so you should add quotes if appropriate for your shell. The wrapper runs until it executes your program, and then GDB takes control.

You can use any program that eventually calls `execve` with its arguments as a wrapper. Several standard Unix utilities do this, e.g. `env` and `nohup`. Any Unix shell script ending with `exec "$@"` will also work.

For example, you can use `env` to pass an environment variable to the debugged program, without setting the variable in your shell's environment:

```
(gdb) set exec-wrapper env 'LD_PRELOAD=libtest.so'
(gdb) run
```

This command is available when debugging locally on most targets, excluding DJGPP, Cygwin, MS Windows, and QNX Neutrino.

`set startup-with-shell`
`set startup-with-shell on`
`set startup-with-shell off`
`show startup-with-shell`

On Unix systems, by default, if a shell is available on your target, GDB) uses it to start your program. Arguments of the `run` command are passed to the shell, which does variable substitution, expands wildcard characters and performs redirection of I/O. In some circumstances, it may be useful to disable such use of a shell, for example, when debugging the shell itself or diagnosing startup failures such as:

```
(gdb) run
Starting program: ./a.out
During startup program terminated with signal SIGSEGV, Segmentation fault.
```

which indicates the shell or the wrapper specified with '`exec-wrapper`' crashed, not your program. Most often, this is caused by something odd in your shell's non-interactive mode initialization file—such as `.cshrc` for C-shell, `$.zshenv` for the Z shell, or the file specified in the `BASH_ENV` environment variable for BASH.

`set auto-connect-native-target`
`set auto-connect-native-target on`
`set auto-connect-native-target off`
`show auto-connect-native-target`

By default, if the current inferior is not connected to any target yet (e.g., with `target remote`), the `run` command starts your program as a native process under GDB, on your local machine. If you're sure you don't want to debug programs on your local machine, you can tell GDB to not connect to the native target automatically with the `set auto-connect-native-target off` command.

If `on`, which is the default, and if the current inferior is not connected to a target already, the `run` command automaticaly connects to the native target, if one is available.

If `off`, and if the current inferior is not connected to a target already, the `run` command fails with an error:

```
(gdb) run
Don't know how to run.  Try "help target".
```

If the current inferior is already connected to a target, GDB always uses it with the `run` command.

In any case, you can explicitly connect to the native target with the `target native` command. For example,

```
(gdb) set auto-connect-native-target off
(gdb) run
Don't know how to run.  Try "help target".
(gdb) target native
(gdb) run
Starting program: ./a.out
[Inferior 1 (process 10421) exited normally]
```

In case you connected explicitly to the `native` target, GDB remains connected even if all inferiors exit, ready for the next `run` command. Use the `disconnect` command to disconnect.

Examples of other commands that likewise respect the `auto-connect-native-target` setting: `attach`, `info proc`, `info os`.

`set disable-randomization`
`set disable-randomization on`

This option (enabled by default in GDB) will turn off the native randomization of the virtual address space of the started program. This option is useful for multiple debugging sessions to make the execution better reproducible and memory addresses reusable across debugging sessions.

This feature is implemented only on certain targets, including GNU/Linux. On GNU/Linux you can get the same behavior using

```
(gdb) set exec-wrapper setarch 'uname -m' -R
```

`set disable-randomization off`

Leave the behavior of the started executable unchanged. Some bugs rear their ugly heads only when the program is loaded at certain addresses. If your bug disappears when you run the program under GDB, that might be because GDB by default disables the address randomization on platforms, such as GNU/Linux, which do that for stand-alone programs. Use *`set disable-randomization off`* to try to reproduce such elusive bugs.

On targets where it is available, virtual address space randomization protects the programs against certain kinds of security attacks. In these cases the attacker needs to know the exact location of a concrete executable code. Randomizing its location makes it impossible to inject jumps misusing a code at its expected addresses.

Prelinking shared libraries provides a startup performance advantage but it makes addresses in these libraries predictable for privileged processes by having just unprivileged access at the target system. Reading the shared library binary gives enough information for assembling the malicious code misusing it. Still even a prelinked shared library can get loaded at a new random address just requiring the regular relocation process during the startup. Shared libraries not already prelinked are always loaded at a randomly chosen address.

Position independent executables (PIE) contain position independent code similar to the shared libraries and therefore such executables get loaded at a randomly chosen address upon startup. PIE executables always load even already

prelinked shared libraries at a random address. You can build such executable using `gcc -fPIE -pie`.

Heap (malloc storage), stack and custom mmap areas are always placed randomly (as long as the randomization is enabled).

`show disable-randomization`
: Show the current setting of the explicit disable of the native randomization of the virtual address space of the started program.

4.3 Your Program's Arguments

The arguments to your program can be specified by the arguments of the `run` command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to your program. Your `SHELL` environment variable (if it exists) specifies what shell GDB uses. If you do not define `SHELL`, GDB uses the default shell (`/bin/sh` on Unix).

On non-Unix systems, the program is usually invoked directly by GDB, which emulates I/O redirection via the appropriate system calls, and the wildcard characters are expanded by the startup code of the program, not by the shell.

`run` with no arguments uses the same arguments used by the previous `run`, or those set by the `set args` command.

`set args` Specify the arguments to be used the next time your program is run. If `set args` has no arguments, `run` executes your program with no arguments. Once you have run your program with arguments, using `set args` before the next `run` is the only way to run it again without arguments.

`show args` Show the arguments to give your program when it is started.

4.4 Your Program's Environment

The *environment* consists of a set of environment variables and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they are inherited by all the other programs you run. When debugging, it can be useful to try running your program with a modified environment without having to start GDB over again.

`path` *directory*
: Add *directory* to the front of the `PATH` environment variable (the search path for executables) that will be passed to your program. The value of `PATH` used by GDB does not change. You may specify several directory names, separated by whitespace or by a system-dependent separator character (‘`:`’ on Unix, ‘`;`’ on MS-DOS and MS-Windows). If *directory* is already in the path, it is moved to the front, so it is searched sooner.

  You can use the string ‘`$cwd`’ to refer to whatever is the current working directory at the time GDB searches the path. If you use ‘`.`’ instead, it refers to the directory where you executed the `path` command. GDB replaces ‘`.`’ in the *directory* argument (with the current path) before adding *directory* to the search path.

`show paths`
: Display the list of search paths for executables (the `PATH` environment variable).

`show environment` [*varname*]
: Print the value of environment variable *varname* to be given to your program when it starts. If you do not supply *varname*, print the names and values of all environment variables to be given to your program. You can abbreviate `environment` as `env`.

`set environment` *varname* [`=`*value*]
: Set environment variable *varname* to *value*. The value changes for your program (and the shell GDB uses to launch it), not for GDB itself. The *value* may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The *value* parameter is optional; if it is eliminated, the variable is set to a null value.

  For example, this command:

  ```
  set env USER = foo
  ```

  tells the debugged program, when subsequently run, that its user is named '`foo`'. (The spaces around '`=`' are used for clarity here; they are not actually required.)

  Note that on Unix systems, GDB runs your program via a shell, which also inherits the environment set with `set environment`. If necessary, you can avoid that by using the '`env`' program as a wrapper instead of using `set environment`. See [set exec-wrapper], page 33, for an example doing just that.

  Environment variables that are set by the user are also transmitted to `gdbserver` to be used when starting the remote inferior. see [QEnvironmentHexEncoded], page 701.

`unset environment` *varname*
: Remove variable *varname* from the environment to be passed to your program. This is different from '`set env` *varname* `=`'; `unset environment` removes the variable from the environment, rather than assigning it an empty value.

  Environment variables that are unset by the user are also unset on `gdbserver` when starting the remote inferior. see [QEnvironmentUnset], page 701.

Warning: On Unix systems, GDB runs your program using the shell indicated by your `SHELL` environment variable if it exists (or `/bin/sh` if not). If your `SHELL` variable names a shell that runs an initialization file when started non-interactively—such as `.cshrc` for C-shell, `$.zshenv` for the Z shell, or the file specified in the `BASH_ENV` environment variable for BASH—any variables you set in that file affect your program. You may wish to move setting of environment variables to files that are only run when you sign on, such as `.login` or `.profile`.

4.5 Your Program's Working Directory

Each time you start your program with `run`, the inferior will be initialized with the current working directory specified by the *`set cwd`* command. If no directory has been specified by this command, then the inferior will inherit GDB's current working directory as its working

directory if native debugging, or it will inherit the remote server's current working directory if remote debugging.

`set cwd` [*directory*]
: Set the inferior's working directory to *directory*, which will be `glob`-expanded in order to resolve tildes (`~`). If no argument has been specified, the command clears the setting and resets it to an empty state. This setting has no effect on GDB's working directory, and it only takes effect the next time you start the inferior. The `~` in *directory* is a short for the *home directory*, usually pointed to by the `HOME` environment variable. On MS-Windows, if `HOME` is not defined, GDB uses the concatenation of `HOMEDRIVE` and `HOMEPATH` as fallback.

  You can also change GDB's current working directory by using the `cd` command. See [cd command], page 38.

`show cwd`
: Show the inferior's working directory. If no directory has been specified by *`set cwd`*, then the default inferior's working directory is the same as GDB's working directory.

`cd` [*directory*]
: Set the GDB working directory to *directory*. If not given, *directory* uses `'~'`.

  The GDB working directory serves as a default for the commands that specify files for GDB to operate on. See Section 18.1 [Commands to Specify Files], page 277. See [set cwd command], page 38.

`pwd`
: Print the GDB working directory.

It is generally impossible to find the current working directory of the process being debugged (since a program can change its directory during its run). If you work on a system where GDB supports the `info proc` command (see Section 21.1.2 [Process Information], page 321), you can use the `info proc` command to find out the current working directory of the debuggee.

4.6 Your Program's Input and Output

By default, the program you run under GDB does input and output to the same terminal that GDB uses. GDB switches the terminal to its own terminal modes to interact with you, but it records the terminal modes your program was using and switches back to them when you continue running your program.

`info terminal`
: Displays information recorded by GDB about the terminal modes your program is using.

You can redirect your program's input and/or output using shell redirection with the `run` command. For example,

```
run > outfile
```

starts your program, diverting its output to the file `outfile`.

Another way to specify where your program should do input and output is with the `tty` command. This command accepts a file name as argument, and causes this file to be

the default for future `run` commands. It also resets the controlling terminal for the child process, for future `run` commands. For example,

```
tty /dev/ttyb
```

directs that processes started with subsequent `run` commands default to do input and output on the terminal `/dev/ttyb` and have that as their controlling terminal.

An explicit redirection in `run` overrides the `tty` command's effect on the input/output device, but not its effect on the controlling terminal.

When you use the `tty` command or redirect input in the `run` command, only the input *for your program* is affected. The input for GDB still comes from your terminal. `tty` is an alias for `set inferior-tty`.

You can use the `show inferior-tty` command to tell GDB to display the name of the terminal that will be used for future runs of your program.

`set inferior-tty [ tty ]`
: Set the tty for the program being debugged to *tty*. Omitting *tty* restores the default behavior, which is to use the same terminal as GDB.

`show inferior-tty`
: Show the current tty for the program being debugged.

4.7 Debugging an Already-running Process

`attach process-id`
: This command attaches to a running process—one that was started outside GDB. (`info files` shows your active targets.) The command takes as argument a process ID. The usual way to find out the *process-id* of a Unix process is with the `ps` utility, or with the '`jobs -l`' shell command.

  `attach` does not repeat if you press `RET` a second time after executing the command.

To use `attach`, your program must be running in an environment which supports processes; for example, `attach` does not work for programs on bare-board targets that lack an operating system. You must also have permission to send the process a signal.

When you use `attach`, the debugger finds the program running in the process first by looking in the current working directory, then (if the program is not found) by using the source file search path (see Section 9.5 [Specifying Source Directories], page 124). You can also use the `file` command to load the program. See Section 18.1 [Commands to Specify Files], page 277.

If the debugger can determine that the executable file running in the process it is attaching to does not match the current exec-file loaded by GDB, the option `exec-file-mismatch` specifies how to handle the mismatch. GDB tries to compare the files by comparing their build IDs (see [build ID], page 286), if available.

`set exec-file-mismatch 'ask|warn|off'`
: Whether to detect mismatch between the current executable file loaded by GDB and the executable file used to start the process. If '`ask`', the default, display a warning and ask the user whether to load the process executable file; if '`warn`', just display a warning; if '`off`', don't attempt to detect a mismatch. If the user

confirms loading the process executable file, then its symbols will be loaded as well.

`show exec-file-mismatch`
: Show the current value of `exec-file-mismatch`.

The first thing GDB does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with `run`. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, you may use the `continue` command after attaching GDB to the process.

`detach`
: When you have finished debugging the attached process, you can use the `detach` command to release it from GDB control. Detaching the process continues its execution. After the `detach` command, that process and GDB become completely independent once more, and you are ready to `attach` another process or start one with `run`. `detach` does not repeat if you press `RET` again after executing the command.

If you exit GDB while you have an attached process, you detach that process. If you use the `run` command, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command (see Section 22.9 [Optional Warnings and Messages], page 357).

4.8 Killing the Child Process

`kill`
: Kill the child process in which your program is running under GDB.

This command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump file while your program is running.

On some operating systems, a program cannot be executed outside GDB while you have breakpoints set on it inside GDB. You can use the `kill` command in this situation to permit running your program outside the debugger.

The `kill` command is also useful if you wish to recompile and relink your program, since on many systems it is impossible to modify an executable file while it is running in a process. In this case, when you next type `run`, GDB notices that the file has changed, and reads the symbol table again (while trying to preserve your current breakpoint settings).

4.9 Debugging Multiple Inferiors Connections and Programs

GDB lets you run and debug multiple programs in a single session. In addition, GDB on some systems may let you run several programs simultaneously (otherwise you have to exit from one before starting another). On some systems GDB may even let you debug several programs simultaneously on different remote systems. In the most general case, you can have multiple threads of execution in each of multiple processes, launched from multiple executables, running on different machines.

GDB represents the state of each program execution with an object called an *inferior*. An inferior typically corresponds to a process, but is more general and applies also to targets that do not have processes. Inferiors may be created before a process runs, and may

be retained after a process exits. Inferiors have unique identifiers that are different from process ids. Usually each inferior will also have its own distinct address space, although some embedded targets may have several inferiors running in different parts of a single address space. Each inferior may in turn have multiple threads running in it.

To find out what inferiors exist at any moment, use `info inferiors`:

`info inferiors`

Print a list of all inferiors currently being managed by GDB. By default all inferiors are printed, but the argument *id*... – a space separated list of inferior numbers – can be used to limit the display to just the requested inferiors.

GDB displays for each inferior (in this order):

1. the inferior number assigned by GDB
2. the target system's inferior identifier
3. the target connection the inferior is bound to, including the unique connection number assigned by GDB, and the protocol used by the connection.
4. the name of the executable the inferior is running.

An asterisk '`*`' preceding the GDB inferior number indicates the current inferior.

For example,

```
(gdb) info inferiors
  Num  Description       Connection                      Executable
* 1    process 3401      1 (native)                      goodbye
  2    process 2307      2 (extended-remote host:10000)  hello
```

To get informations about the current inferior, use `inferior`:

`inferior` Shows information about the current inferior.

For example,

```
(gdb) inferior
[Current inferior is 1 [process 3401] (helloworld)]
```

To find out what open target connections exist at any moment, use `info connections`:

`info connections`

Print a list of all open target connections currently being managed by GDB. By default all connections are printed, but the argument *id*... – a space separated list of connections numbers – can be used to limit the display to just the requested connections.

GDB displays for each connection (in this order):

1. the connection number assigned by GDB.
2. the protocol used by the connection.
3. a textual description of the protocol used by the connection.

An asterisk '`*`' preceding the connection number indicates the connection of the current inferior.

For example,

```
(gdb) info connections
  Num  What                          Description
* 1    extended-remote host:10000  Extended remote serial target in gdb-specific protocol
  2    native                      Native process
```

```
  3    core                          Local core dump file
```

To switch focus between inferiors, use the `inferior` command:

`inferior` *infno*
: Make inferior number *infno* the current inferior. The argument *infno* is the inferior number assigned by GDB, as shown in the first field of the '`info inferiors`' display.

The debugger convenience variable '`$_inferior`' contains the number of the current inferior. You may find this useful in writing breakpoint conditional expressions, command scripts, and so forth. See Section 10.12 [Convenience Variables], page 162, for general information on convenience variables.

You can get multiple executables into a debugging session via the `add-inferior` and `clone-inferior` commands. On some systems GDB can add inferiors to the debug session automatically by following calls to `fork` and `exec`. To remove inferiors from the debugging session use the `remove-inferiors` command.

`add-inferior [ -copies` *n* `] [ -exec` *executable* `] [-no-connection ]`
: Adds *n* inferiors to be run using *executable* as the executable; *n* defaults to 1. If no executable is specified, the inferiors begins empty, with no program. You can still assign or change the program assigned to the inferior at any time by using the `file` command with the executable name as its argument.

  By default, the new inferior begins connected to the same target connection as the current inferior. For example, if the current inferior was connected to `gdbserver` with `target remote`, then the new inferior will be connected to the same `gdbserver` instance. The '`-no-connection`' option starts the new inferior with no connection yet. You can then for example use the `target remote` command to connect to some other `gdbserver` instance, use `run` to spawn a local program, etc.

`clone-inferior [ -copies` *n* `] [` *infno* `]`
: Adds *n* inferiors ready to execute the same program as inferior *infno*; *n* defaults to 1, and *infno* defaults to the number of the current inferior. This command copies the values of the *args*, *inferior-tty* and *cwd* properties from the current inferior to the new one. It also propagates changes the user made to environment variables using the `set environment` and `unset environment` commands. This is a convenient command when you want to run another instance of the inferior you are debugging.

```
(gdb) info inferiors
  Num  Description       Connection   Executable
* 1    process 29964     1 (native)   helloworld
(gdb) clone-inferior
Added inferior 2.
1 inferiors added.
(gdb) info inferiors
  Num  Description       Connection   Executable
* 1    process 29964     1 (native)   helloworld
  2    <null>            1 (native)   helloworld
```

You can now simply switch focus to inferior 2 and run it.

`remove-inferiors` *infno*`...`
: Removes the inferior or inferiors *infno*.... It is not possible to remove an inferior that is running with this command. For those, use the `kill` or `detach` command first.

To quit debugging one of the running inferiors that is not the current inferior, you can either detach from it by using the `detach inferior` command (allowing it to run independently), or kill it using the `kill inferiors` command:

`detach inferior` *infno*`...`
: Detach from the inferior or inferiors identified by GDB inferior number(s) *infno*.... Note that the inferior's entry still stays on the list of inferiors shown by `info inferiors`, but its Description will show '`<null>`'.

`kill inferiors` *infno*`...`
: Kill the inferior or inferiors identified by GDB inferior number(s) *infno*.... Note that the inferior's entry still stays on the list of inferiors shown by `info inferiors`, but its Description will show '`<null>`'.

After the successful completion of a command such as `detach`, `detach inferiors`, `kill` or `kill inferiors`, or after a normal process exit, the inferior is still valid and listed with `info inferiors`, ready to be restarted.

To be notified when inferiors are started or exit under GDB's control use `set print inferior-events`:

`set print inferior-events`
`set print inferior-events on`
`set print inferior-events off`
: The `set print inferior-events` command allows you to enable or disable printing of messages when GDB notices that new inferiors have started or that inferiors have exited or have been detached. By default, these messages will be printed.

`show print inferior-events`
: Show whether messages will be printed when GDB detects that inferiors have started, exited or have been detached.

Many commands will work the same with multiple programs as with a single program: e.g., `print myglobal` will simply display the value of `myglobal` in the current inferior.

Occasionally, when debugging GDB itself, it may be useful to get more info about the relationship of inferiors, programs, address spaces in a debug session. You can do that with the `maint info program-spaces` command.

`maint info program-spaces`
: Print a list of all program spaces currently being managed by GDB.

  GDB displays for each program space (in this order):

  1. the program space number assigned by GDB
  2. the name of the executable loaded into the program space, with e.g., the `file` command.

An asterisk '`*`' preceding the GDB program space number indicates the current program space.

In addition, below each program space line, GDB prints extra information that isn't suitable to display in tabular form. For example, the list of inferiors bound to the program space.

```
(gdb) maint info program-spaces
  Id   Executable
* 1    hello
  2    goodbye
       Bound inferiors: ID 1 (process 21561)
```

Here we can see that no inferior is running the program `hello`, while `process 21561` is running the program `goodbye`. On some targets, it is possible that multiple inferiors are bound to the same program space. The most common example is that of debugging both the parent and child processes of a `vfork` call. For example,

```
(gdb) maint info program-spaces
  Id   Executable
* 1    vfork-test
       Bound inferiors: ID 2 (process 18050), ID 1 (process 18045)
```

Here, both inferior 2 and inferior 1 are running in the same program space as a result of inferior 1 having executed a `vfork` call.

4.10 Debugging Programs with Multiple Threads

In some operating systems, such as GNU/Linux and Solaris, a single program may have more than one *thread* of execution. The precise semantics of threads differ from one operating system to another, but in general the threads of a single program are akin to multiple processes—except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and perhaps private memory.

GDB provides these facilities for debugging multi-thread programs:

- automatic notification of new threads
- '`thread` *`thread-id`*', a command to switch among threads
- '`info threads`', a command to inquire about existing threads
- '`thread apply [`*`thread-id-list`*` | all]` *`args`*', a command to apply a command to a list of threads
- thread-specific breakpoints
- '`set print thread-events`', which controls printing of messages on thread start and exit.
- '`set libthread-db-search-path` *`path`*', which lets the user specify which `libthread_db` to use if the default choice isn't compatible with the program.

The GDB thread debugging facility allows you to observe all threads while your program runs—but whenever GDB takes control, one thread in particular is always the focus of debugging. This thread is called the *current thread*. Debugging commands show program information from the perspective of the current thread.

Whenever GDB detects a new thread in your program, it displays the target system's identification for the thread with a message in the form '`[New systag]`', where *systag* is a thread identifier whose form varies depending on the particular system. For example, on GNU/Linux, you might see

```
[New Thread 0x41e02940 (LWP 25582)]
```

when GDB notices a new thread. In contrast, on other systems, the *systag* is simply something like '`process 368`', with no further qualifier.

For debugging purposes, GDB associates its own thread number —always a single integer—with each thread of an inferior. This number is unique between all threads of an inferior, but not unique between threads of different inferiors.

You can refer to a given thread in an inferior using the qualified *inferior-num.thread-num* syntax, also known as *qualified thread ID*, with *inferior-num* being the inferior number and *thread-num* being the thread number of the given inferior. For example, thread `2.3` refers to thread number 3 of inferior 2. If you omit *inferior-num* (e.g., `thread 3`), then GDB infers you're referring to a thread of the current inferior.

Until you create a second inferior, GDB does not show the *inferior-num* part of thread IDs, even though you can always use the full *inferior-num.thread-num* form to refer to threads of inferior 1, the initial inferior.

Some commands accept a space-separated *thread ID list* as argument. A list element can be:

1. A thread ID as shown in the first field of the '`info threads`' display, with or without an inferior qualifier. E.g., '`2.1`' or '`1`'.
2. A range of thread numbers, again with or without an inferior qualifier, as in *inf.thr1-thr2* or *thr1-thr2*. E.g., '`1.2-4`' or '`2-4`'.
3. All threads of an inferior, specified with a star wildcard, with or without an inferior qualifier, as in *inf.*`*` (e.g., '`1.*`') or `*`. The former refers to all threads of the given inferior, and the latter form without an inferior qualifier refers to all threads of the current inferior.

For example, if the current inferior is 1, and inferior 7 has one thread with ID 7.1, the thread list '`1 2-3 4.5 6.7-9 7.*`' includes threads 1 to 3 of inferior 1, thread 5 of inferior 4, threads 7 to 9 of inferior 6 and all threads of inferior 7. That is, in expanded qualified form, the same as '`1.1 1.2 1.3 4.5 6.7 6.8 6.9 7.1`'.

In addition to a *per-inferior* number, each thread is also assigned a unique *global* number, also known as *global thread ID*, a single integer. Unlike the thread number component of the thread ID, no two threads have the same global ID, even when you're debugging multiple inferiors.

From GDB's perspective, a process always has at least one thread. In other words, GDB assigns a thread number to the program's "main thread" even if the program is not multi-threaded.

The debugger convenience variables '`$_thread`' and '`$_gthread`' contain, respectively, the per-inferior thread number and the global thread number of the current thread. You may find this useful in writing breakpoint conditional expressions, command scripts, and so forth. See Section 10.12 [Convenience Variables], page 162, for general information on convenience variables.

If GDB detects the program is multi-threaded, it augments the usual message about stopping at a breakpoint with the ID and name of the thread that hit the breakpoint.

```
Thread 2 "client" hit Breakpoint 1, send_message () at client.c:68
```

Likewise when the program receives a signal:

```
Thread 1 "main" received signal SIGINT, Interrupt.
```

`info threads` [*thread-id-list*]

Display information about one or more threads. With no arguments displays information about all threads. You can specify the list of threads that you want to display using the thread ID list syntax (see [thread ID lists], page 45).

GDB displays for each thread (in this order):

1. the per-inferior thread number assigned by GDB
2. the global thread number assigned by GDB, if the '`-gid`' option was specified
3. the target system's thread identifier (*systag*)
4. the thread's name, if one is known. A thread can either be named by the user (see `thread name`, below), or, in some cases, by the program itself.
5. the current stack frame summary for that thread

An asterisk '`*`' to the left of the GDB thread number indicates the current thread.

For example,

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

If you're debugging multiple inferiors, GDB displays thread IDs using the qualified *inferior-num.thread-num* format. Otherwise, only *thread-num* is shown.

If you specify the '`-gid`' option, GDB displays a column indicating each thread's global thread ID:

```
(gdb) info threads
  Id   GId  Target Id             Frame
  1.1  1    process 35 thread 13  main (argc=1, argv=0x7ffffff8)
  1.2  3    process 35 thread 23  0x34e5 in sigpause ()
  1.3  4    process 35 thread 27  0x34e5 in sigpause ()
* 2.1  2    process 65 thread 1   main (argc=1, argv=0x7ffffff8)
```

On Solaris, you can display more information about user threads with a Solaris-specific command:

`maint info sol-threads`

Display info on Solaris user threads.

`thread` *thread-id*

Make thread ID *thread-id* the current thread. The command argument *thread-id* is the GDB thread ID, as shown in the first field of the '`info threads`' display, with or without an inferior qualifier (e.g., '`2.1`' or '`1`').

GDB responds by displaying the system identifier of the thread you selected, and its current stack frame summary:

```
(gdb) thread 2
```

```
[Switching to thread 2 (Thread 0xb7fdab70 (LWP 12747))]
#0  some_function (ignore=0x0) at example.c:8
8     printf ("hello\n");
```

As with the '`[New ...]`' message, the form of the text after '`Switching to`' depends on your system's conventions for identifying threads.

`thread apply [`*`thread-id-list`*` | all [-ascending]] [`*`flag`*`]... `*`command`*

The `thread apply` command allows you to apply the named *command* to one or more threads. Specify the threads that you want affected using the thread ID list syntax (see [thread ID lists], page 45), or specify `all` to apply to all threads. To apply a command to all threads in descending order, type *`thread apply all command`*. To apply a command to all threads in ascending order, type *`thread apply all -ascending command`*.

The *flag* arguments control what output to produce and how to handle errors raised when applying *command* to a thread. *flag* must start with a `-` directly followed by one letter in `qcs`. If several flags are provided, they must be given individually, such as `-c -q`.

By default, GDB displays some thread information before the output produced by *command*, and an error raised during the execution of a *command* will abort `thread apply`. The following flags can be used to fine-tune this behavior:

`-c` The flag `-c`, which stands for '`continue`', causes any errors in *command* to be displayed, and the execution of `thread apply` then continues.

`-s` The flag `-s`, which stands for '`silent`', causes any errors or empty output produced by a *command* to be silently ignored. That is, the execution continues, but the thread information and errors are not printed.

`-q` The flag `-q` ('`quiet`') disables printing the thread information.

Flags `-c` and `-s` cannot be used together.

`taas [`*`option`*`]... `*`command`*

Shortcut for `thread apply all -s [`*`option`*`]... `*`command`*. Applies *command* on all threads, ignoring errors and empty output.

The `taas` command accepts the same options as the `thread apply all` command. See [thread apply all], page 47.

`tfaas [`*`option`*`]... `*`command`*

Shortcut for `thread apply all -s -- frame apply all -s [`*`option`*`]... `*`command`*. Applies *command* on all frames of all threads, ignoring errors and empty output. Note that the flag `-s` is specified twice: The first `-s` ensures that `thread apply` only shows the thread information of the threads for which `frame apply` produces some output. The second `-s` is needed to ensure that `frame apply` shows the frame information of a frame only if the *command* successfully produced some output.

It can for example be used to print a local variable or a function argument without knowing the thread or frame where this variable or argument is, using:

```
(gdb) tfaas p some_local_var_i_do_not_remember_where_it_is
```

The `tfaas` command accepts the same options as the `frame apply` command. See Section 8.5 [frame apply], page 114.

`thread name [name]`
: This command assigns a name to the current thread. If no argument is given, any existing user-specified name is removed. The thread name appears in the '`info threads`' display.

  On some systems, such as GNU/Linux, GDB is able to determine the name of the thread as given by the OS. On these systems, a name specified with '`thread name`' will override the system-give name, and removing the user-specified name will cause GDB to once again display the system-specified name.

`thread find [regexp]`
: Search for and display thread ids whose name or *systag* matches the supplied regular expression.

  As well as being the complement to the '`thread name`' command, this command also allows you to identify a thread by its target *systag*. For instance, on GNU/Linux, the target *systag* is the LWP id.

```
(GDB) thread find 26688
Thread 4 has target id 'Thread 0x41e02940 (LWP 26688)'
(GDB) info thread 4
  Id   Target Id         Frame
  4    Thread 0x41e02940 (LWP 26688) 0x00000031ca6cd372 in select ()
```

`set print thread-events`
`set print thread-events on`
`set print thread-events off`
: The `set print thread-events` command allows you to enable or disable printing of messages when GDB notices that new threads have started or that threads have exited. By default, these messages will be printed if detection of these events is supported by the target. Note that these messages cannot be disabled on all targets.

`show print thread-events`
: Show whether messages will be printed when GDB detects that threads have started and exited.

See Section 5.5 [Stopping and Starting Multi-thread Programs], page 89, for more information about how GDB behaves when you stop and start programs with multiple threads.

See Section 5.1.2 [Setting Watchpoints], page 63, for information about watchpoints in programs with multiple threads.

`set libthread-db-search-path [path]`
: If this variable is set, *path* is a colon-separated list of directories GDB will use to search for `libthread_db`. If you omit *path*, '`libthread-db-search-path`' will be reset to its default value (`$sdir:$pdir` on GNU/Linux and Solaris systems). Internally, the default value comes from the `LIBTHREAD_DB_SEARCH_PATH` macro.

  On GNU/Linux and Solaris systems, GDB uses a "helper" `libthread_db` library to obtain information about threads in the inferior process. GDB will use

'`libthread-db-search-path`' to find `libthread_db`. GDB also consults first if inferior specific thread debugging library loading is enabled by '`set auto-load libthread-db`' (see Section 22.8.2 [libthread_db.so.1 file], page 355).

A special entry '`$sdir`' for '`libthread-db-search-path`' refers to the default system directories that are normally searched for loading shared libraries. The '`$sdir`' entry is the only kind not needing to be enabled by '`set auto-load libthread-db`' (see Section 22.8.2 [libthread_db.so.1 file], page 355).

A special entry '`$pdir`' for '`libthread-db-search-path`' refers to the directory from which `libpthread` was loaded in the inferior process.

For any `libthread_db` library GDB finds in above directories, GDB attempts to initialize it with the current inferior process. If this initialization fails (which could happen because of a version mismatch between `libthread_db` and `libpthread`), GDB will unload `libthread_db`, and continue with the next directory. If none of `libthread_db` libraries initialize successfully, GDB will issue a warning and thread debugging will be disabled.

Setting `libthread-db-search-path` is currently implemented only on some platforms.

`show libthread-db-search-path`
: Display current libthread_db search path.

`set debug libthread-db`
`show debug libthread-db`
: Turns on or off display of `libthread_db`-related events. Use `1` to enable, `0` to disable.

`set debug threads` [`on`|`off`]
`show debug threads`
: When '`on`' GDB will print additional messages when threads are created and deleted.

4.11 Debugging Forks

On most systems, GDB has no special support for debugging programs which create additional processes using the `fork` function. When a program forks, GDB will continue to debug the parent process and the child process will run unimpeded. If you have set a breakpoint in any code which the child then executes, the child will get a `SIGTRAP` signal which (unless it catches the signal) will cause it to terminate.

However, if you want to debug the child process there is a workaround which isn't too painful. Put a call to `sleep` in the code which the child process executes after the fork. It may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the `ps` program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process (see Section 4.7 [Attach], page 39). From that point on you can debug the child process just like any other process which you attached to.

On some systems, GDB provides support for debugging programs that create additional processes using the `fork` or `vfork` functions. On GNU/Linux platforms, this feature is supported with kernel version 2.5.46 and later.

The fork debugging commands are supported in native mode and when connected to `gdbserver` in either `target remote` mode or `target extended-remote` mode.

By default, when a program forks, GDB will continue to debug the parent process and the child process will run unimpeded.

If you want to follow the child process instead of the parent process, use the command `set follow-fork-mode`.

`set follow-fork-mode` *mode*

Set the debugger response to a program call of `fork` or `vfork`. A call to `fork` or `vfork` creates a new process. The *mode* argument can be:

`parent` The original process is debugged after a fork. The child process runs unimpeded. This is the default.

`child` The new process is debugged after a fork. The parent process runs unimpeded.

`show follow-fork-mode`

Display the current debugger response to a `fork` or `vfork` call.

On Linux, if you want to debug both the parent and child processes, use the command `set detach-on-fork`.

`set detach-on-fork` *mode*

Tells gdb whether to detach one of the processes after a fork, or retain debugger control over them both.

`on` The child process (or parent process, depending on the value of `follow-fork-mode`) will be detached and allowed to run independently. This is the default.

`off` Both processes will be held under the control of GDB. One process (child or parent, depending on the value of `follow-fork-mode`) is debugged as usual, while the other is held suspended.

`show detach-on-fork`

Show whether detach-on-fork mode is on/off.

If you choose to set '`detach-on-fork`' mode off, then GDB will retain control of all forked processes (including nested forks). You can list the forked processes under the control of GDB by using the `info inferiors` command, and switch from one fork to another by using the `inferior` command (see Section 4.9 [Debugging Multiple Inferiors Connections and Programs], page 40).

To quit debugging one of the forked processes, you can either detach from it by using the `detach inferiors` command (allowing it to run independently), or kill it using the `kill inferiors` command. See Section 4.9 [Debugging Multiple Inferiors Connections and Programs], page 40.

If you ask to debug a child process and a `vfork` is followed by an `exec`, GDB executes the new target up to the first breakpoint in the new target. If you have a breakpoint set on `main` in your original program, the breakpoint will also be set on the child process's `main`.

On some systems, when a child process is spawned by `vfork`, you cannot debug the child or parent until an `exec` call completes.

If you issue a `run` command to GDB after an `exec` call executes, the new target restarts. To restart the parent process, use the `file` command with the parent executable name as its argument. By default, after an `exec` call executes, GDB discards the symbols of the previous executable image. You can change this behaviour with the `set follow-exec-mode` command.

`set follow-exec-mode` *mode*

Set debugger response to a program call of `exec`. An `exec` call replaces the program image of a process.

`follow-exec-mode` can be:

`new` GDB creates a new inferior and rebinds the process to this new inferior. The program the process was running before the `exec` call can be restarted afterwards by restarting the original inferior.

For example:

```
(gdb) info inferiors
(gdb) info inferior
  Id   Description   Executable
* 1    <null>        prog1
(gdb) run
process 12020 is executing new program: prog2
Program exited normally.
(gdb) info inferiors
  Id   Description   Executable
  1    <null>        prog1
* 2    <null>        prog2
```

`same` GDB keeps the process bound to the same inferior. The new executable image replaces the previous executable loaded in the inferior. Restarting the inferior after the `exec` call, with e.g., the `run` command, restarts the executable the process was running after the `exec` call. This is the default mode.

For example:

```
(gdb) info inferiors
  Id   Description   Executable
* 1    <null>        prog1
(gdb) run
process 12020 is executing new program: prog2
Program exited normally.
(gdb) info inferiors
  Id   Description   Executable
* 1    <null>        prog2
```

`follow-exec-mode` is supported in native mode and `target extended-remote` mode.

You can use the `catch` command to make GDB stop whenever a `fork`, `vfork`, or `exec` call is made. See Section 5.1.3 [Setting Catchpoints], page 66.

4.12 Setting a *Bookmark* to Return to Later

On certain operating systems[1], GDB is able to save a *snapshot* of a program's state, called a *checkpoint*, and come back to it later.

Returning to a checkpoint effectively undoes everything that has happened in the program since the `checkpoint` was saved. This includes changes in memory, registers, and even (within some limits) system state. Effectively, it is like going back in time to the moment when the checkpoint was saved.

Thus, if you're stepping thru a program and you think you're getting close to the point where things go wrong, you can save a checkpoint. Then, if you accidentally go too far and miss the critical statement, instead of having to restart your program from the beginning, you can just go back to the checkpoint and start again from there.

This can be especially useful if it takes a lot of time or steps to reach the point where you think the bug occurs.

To use the `checkpoint`/`restart` method of debugging:

`checkpoint`
: Save a snapshot of the debugged program's current execution state. The `checkpoint` command takes no arguments, but each checkpoint is assigned a small integer id, similar to a breakpoint id.

`info checkpoints`
: List the checkpoints that have been saved in the current debugging session. For each checkpoint, the following information will be listed:

  ```
  Checkpoint ID
  Process ID
  Code Address
  Source line, or label
  ```

`restart` *`checkpoint-id`*
: Restore the program state that was saved as checkpoint number *checkpoint-id*. All program variables, registers, stack frames etc. will be returned to the values that they had when the checkpoint was saved. In essence, gdb will "wind back the clock" to the point in time when the checkpoint was saved.

  Note that breakpoints, GDB variables, command history etc. are not affected by restoring a checkpoint. In general, a checkpoint only restores things that reside in the program being debugged, not in the debugger.

`delete checkpoint` *`checkpoint-id`*
: Delete the previously-saved checkpoint identified by *checkpoint-id*.

Returning to a previously saved checkpoint will restore the user state of the program being debugged, plus a significant subset of the system (OS) state, including file pointers. It won't "un-write" data from a file, but it will rewind the file pointer to the previous location, so that the previously written data can be overwritten. For files opened in read mode, the pointer will also be restored so that the previously read data can be read again.

[1] Currently, only GNU/Linux.

Of course, characters that have been sent to a printer (or other external device) cannot be "snatched back", and characters received from eg. a serial device can be removed from internal program buffers, but they cannot be "pushed back" into the serial pipeline, ready to be received again. Similarly, the actual contents of files that have been changed cannot be restored (at this time).

However, within those constraints, you actually can "rewind" your program to a previously saved point in time, and begin debugging it again — and you can change the course of events so as to debug a different execution path this time.

Finally, there is one bit of internal program state that will be different when you return to a checkpoint — the program's process id. Each checkpoint will have a unique process id (or *pid*), and each will be different from the program's original *pid*. If your program has saved a local copy of its process id, this could potentially pose a problem.

4.12.1 A Non-obvious Benefit of Using Checkpoints

On some systems such as GNU/Linux, address space randomization is performed on new processes for security reasons. This makes it difficult or impossible to set a breakpoint, or watchpoint, on an absolute address if you have to restart the program, since the absolute location of a symbol will change from one execution to the next.

A checkpoint, however, is an *identical* copy of a process. Therefore if you create a checkpoint at (eg.) the start of main, and simply return to that checkpoint instead of restarting the process, you can avoid the effects of address randomization and your symbols will all stay in the same place.

5 Stopping and Continuing

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and find out why.

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. Usually, the messages shown by GDB provide ample explanation of the status of your program—but you can also explicitly request this information at any time.

`info program`
: Display information about the status of your program: whether it is running or not, what process it is, and why it stopped.

5.1 Breakpoints, Watchpoints, and Catchpoints

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the `break` command and its variants (see Section 5.1.1 [Setting Breakpoints], page 56), to specify the place where your program should stop by line number, function name or exact address in the program.

On some systems, you can set breakpoints in shared libraries before the executable is run.

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. The expression may be a value of a variable, or it could involve values of one or more variables combined by operators, such as '`a + b`'. This is sometimes called *data breakpoints*. You must use a different command to set watchpoints (see Section 5.1.2 [Setting Watchpoints], page 63), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See Section 10.8 [Automatic Display], page 147.

A *catchpoint* is another special breakpoint that stops your program when a certain kind of event occurs, such as the throwing of a C++ exception or the loading of a library. As with watchpoints, you use a different command to set a catchpoint (see Section 5.1.3 [Setting Catchpoints], page 66), but aside from that, you can manage a catchpoint like any other breakpoint. (To stop when your program receives a signal, use the `handle` command; see Section 5.4 [Signals], page 86.)

GDB assigns a number to each breakpoint, watchpoint, or catchpoint when you create it; these numbers are successive integers starting with one. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be *enabled* or *disabled*; if disabled, it has no effect on your program until you enable it again.

Some GDB commands accept a space-separated list of breakpoints on which to operate. A list element can be either a single breakpoint number, like '`5`', or a range of such numbers,

like '5-7'. When a breakpoint list is given to a command, all breakpoints in that list are operated on.

5.1.1 Setting Breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable '`$bpnum`' records the number of the breakpoint you've set most recently; see Section 10.12 [Convenience Variables], page 162, for a discussion of what you can do with convenience variables.

`break` *location*

Set a breakpoint at the given *location*, which can specify a function name, a line number, or an address of an instruction. (See Section 9.2 [Specify Location], page 120, for a list of all the possible ways to specify a *location*.) The breakpoint will stop your program just before it executes any of the code in the specified *location*.

When using source languages that permit overloading of symbols, such as C++, a function name may refer to more than one possible place to break. See Section 10.2 [Ambiguous Expressions], page 137, for a discussion of that situation.

It is also possible to insert a breakpoint that will stop the program only if a specific thread (see Section 5.5.4 [Thread-Specific Breakpoints], page 93) or a specific task (see Section 15.4.10.7 [Ada Tasks], page 243) hits that breakpoint.

`break`

When called without any arguments, `break` sets a breakpoint at the next instruction to be executed in the selected stack frame (see Chapter 8 [Examining the Stack], page 107). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a `finish` command in the frame inside the selected frame—except that `finish` does not leave an active breakpoint. If you use `break` without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops.

GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when your program stopped.

`break ... if` *cond*

Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. '`...`' stands for one of the possible arguments described above (or no argument) specifying where to break. See Section 5.1.6 [Break Conditions], page 72, for more information on breakpoint conditions.

The breakpoint may be mapped to multiple locations. If the breakpoint condition *cond* is invalid at some but not all of the locations, the locations for which the condition is invalid are disabled. For example, GDB reports below that two of the three locations are disabled.

```
(gdb) break func if a == 10
warning: failed to validate condition at location 0x11ce, disabling:
```

```
  No symbol "a" in current context.
warning: failed to validate condition at location 0x11b6, disabling:
  No symbol "a" in current context.
Breakpoint 1 at 0x11b6: func. (3 locations)
```

Locations that are disabled because of the condition are denoted by an uppercase `N` in the output of the `info breakpoints` command:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   <MULTIPLE>
        stop only if a == 10
1.1                         N*  0x00000000000011b6 in ...
1.2                         y   0x00000000000011c2 in ...
1.3                         N*  0x00000000000011ce in ...
(*): Breakpoint condition is invalid at this location.
```

If the breakpoint condition *cond* is invalid in the context of *all* the locations of the breakpoint, GDB refuses to define the breakpoint. For example, if variable `foo` is an undefined variable:

```
(gdb) break func if foo
No symbol "foo" in current context.
```

`break ... -force-condition if` *cond*

There may be cases where the condition *cond* is invalid at all the current locations, but the user knows that it will be valid at a future location; for example, because of a library load. In such cases, by using the `-force-condition` keyword before '`if`', GDB can be forced to define the breakpoint with the given condition expression instead of refusing it.

```
(gdb) break func -force-condition if foo
warning: failed to validate condition at location 1, disabling:
  No symbol "foo" in current context.
warning: failed to validate condition at location 2, disabling:
  No symbol "foo" in current context.
warning: failed to validate condition at location 3, disabling:
  No symbol "foo" in current context.
Breakpoint 1 at 0x1158: test.c:18. (3 locations)
```

This causes all the present locations where the breakpoint would otherwise be inserted, to be disabled, as seen in the example above. However, if there exist locations at which the condition is valid, the `-force-condition` keyword has no effect.

`tbreak` *args*

Set a breakpoint enabled only for one stop. The *args* are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See Section 5.1.5 [Disabling Breakpoints], page 71.

`hbreak` *args*

Set a hardware-assisted breakpoint. The *args* are the same as for the `break` command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. This can be

used with the new trap-generation provided by SPARClite DSU and most x86-based targets. These targets will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. However the hardware breakpoint registers can take a limited number of breakpoints. For example, on the DSU, only two data breakpoints can be set at a time, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones (see Section 5.1.5 [Disabling Breakpoints], page 71). See Section 5.1.6 [Break Conditions], page 72. For remote targets, you can restrict the number of hardware breakpoints GDB will use, see [set remote hardware-breakpoint-limit], page 310.

`thbreak` *args*

Set a hardware-assisted breakpoint enabled only for one stop. The *args* are the same as for the `hbreak` command and the breakpoint is set in the same way. However, like the `tbreak` command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the `hbreak` command, the breakpoint requires hardware support and some target hardware may not have this support. See Section 5.1.5 [Disabling Breakpoints], page 71. See also Section 5.1.6 [Break Conditions], page 72.

`rbreak` *regex*

Set breakpoints on all functions matching the regular expression *regex*. This command sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they are treated just like the breakpoints set with the `break` command. You can delete them, disable them, or make them conditional the same way as any other breakpoint.

In programs using different languages, GDB chooses the syntax to print the list of all breakpoints it sets according to the '`set language`' value: using '`set language auto`' (see Section 15.1.3 [Set Language Automatically], page 218) means to use the language of the breakpoint's function, other values mean to use the manually specified language (see Section 15.1.2 [Set Language Manually], page 218).

The syntax of the regular expression is the standard one used with tools like `grep`. Note that this is different from the syntax used by shells, so for instance `foo*` matches all functions that include an `fo` followed by zero or more `o`s. There is an implicit `.*` leading and trailing the regular expression you supply, so to match only functions that begin with `foo`, use `^foo`.

When debugging C++ programs, `rbreak` is useful for setting breakpoints on overloaded functions that are not members of any special classes.

The `rbreak` command can be used to set breakpoints in **all** the functions in a program, like this:

```
(gdb) rbreak .
```

`rbreak` *file*`:`*regex*

If `rbreak` is called with a filename qualification, it limits the search for functions matching the given regular expression to the specified *file*. This can be used, for example, to set breakpoints on every function in a given file:

```
(gdb) rbreak file.c:.
```

The colon separating the filename qualifier from the regex may optionally be surrounded by spaces.

`info breakpoints` [*list...*]
`info break` [*list...*]

Print a table of all breakpoints, watchpoints, and catchpoints set and not deleted. Optional argument *n* means print information only about the specified breakpoint(s) (or watchpoint(s) or catchpoint(s)). For each breakpoint, following columns are printed:

Breakpoint Numbers

Type Breakpoint, watchpoint, or catchpoint.

Disposition
Whether the breakpoint is marked to be disabled or deleted when hit.

Enabled or Disabled
Enabled breakpoints are marked with '`y`'. '`n`' marks breakpoints that are not enabled.

Address Where the breakpoint is in your program, as a memory address. For a pending breakpoint whose address is not yet known, this field will contain '`<PENDING>`'. Such breakpoint won't fire until a shared library that has the symbol or line referred by breakpoint is loaded. See below for details. A breakpoint with several locations will have '`<MULTIPLE>`' in this field—see below for details.

What Where the breakpoint is in the source for your program, as a file and line number. For a pending breakpoint, the original string passed to the breakpoint command will be listed as it cannot be resolved until the appropriate shared library is loaded in the future.

If a breakpoint is conditional, there are two evaluation modes: "host" and "target". If mode is "host", breakpoint condition evaluation is done by GDB on the host's side. If it is "target", then the condition is evaluated by the target. The `info break` command shows the condition on the line following the affected breakpoint, together with its condition evaluation mode in between parentheses.

Breakpoint commands, if any, are listed after that. A pending breakpoint is allowed to have a condition specified for it. The condition is not parsed for validity until a shared library is loaded that allows the pending breakpoint to resolve to a valid location.

`info break` with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the `x` command are set to the address of the last breakpoint listed (see Section 10.6 [Examining Memory], page 143).

`info break` displays a count of the number of times the breakpoint has been hit. This is especially useful in conjunction with the `ignore` command. You can ignore a large number of breakpoint hits, look at the breakpoint info to see how many times the breakpoint was hit, and then run again, ignoring one less than that number. This will get you quickly to the last hit of that breakpoint.

For a breakpoints with an enable count (xref) greater than 1, `info break` also displays that count.

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see Section 5.1.6 [Break Conditions], page 72).

It is possible that a breakpoint corresponds to several locations in your program. Examples of this situation are:

- Multiple functions in the program may have the same name.
- For a C++ constructor, the GCC compiler generates several instances of the function body, used in different cases.
- For a C++ template function, a given line in the function can correspond to any number of instantiations.
- For an inlined function, a given source line can correspond to several places where that function is inlined.

In all those cases, GDB will insert a breakpoint at all the relevant locations.

A breakpoint with multiple locations is displayed in the breakpoint table using several rows—one header row, followed by one row for each breakpoint location. The header row has '`<MULTIPLE>`' in the address column. The rows for individual locations contain the actual addresses for locations, and show the functions to which those locations belong. The number column for a location is of the form *breakpoint-number.location-number*.

For example:

```
Num     Type           Disp Enb  Address    What
1       breakpoint     keep y    <MULTIPLE>
        stop only if i==1
        breakpoint already hit 1 time
1.1                    y    0x080486a2 in void foo<int>() at t.cc:8
1.2                    y    0x080486ca in void foo<double>() at t.cc:8
```

You cannot delete the individual locations from a breakpoint. However, each location can be individually enabled or disabled by passing *breakpoint-number.location-number* as argument to the `enable` and `disable` commands. It's also possible to `enable` and `disable` a range of *location-number* locations using a *breakpoint-number* and two *location-number*s, in increasing order, separated by a hyphen, like *`breakpoint-number.location-number1-location-number2`*, in which case GDB acts on all the locations in the range (inclusive). Disabling or enabling the parent breakpoint (see Section 5.1.5 [Disabling], page 71) affects all of the locations that belong to that breakpoint.

It's quite common to have a breakpoint inside a shared library. Shared libraries can be loaded and unloaded explicitly, and possibly repeatedly, as the program is executed. To support this use case, GDB updates breakpoint locations whenever any shared library is loaded or unloaded. Typically, you would set a breakpoint in a shared library at the beginning of your debugging session, when the library is not loaded, and when the symbols from the library are not available. When you try to set breakpoint, GDB will ask you if you want to set a so called *pending breakpoint*—breakpoint whose address is not yet resolved.

After the program is run, whenever a new shared library is loaded, GDB reevaluates all the breakpoints. When a newly loaded shared library contains the symbol or line referred to by some pending breakpoint, that breakpoint is resolved and becomes an ordinary breakpoint.

When a library is unloaded, all breakpoints that refer to its symbols or source lines become pending again.

This logic works for breakpoints with multiple locations, too. For example, if you have a breakpoint in a C++ template function, and a newly loaded shared library has an instantiation of that template, a new location is added to the list of locations for the breakpoint.

Except for having unresolved address, pending breakpoints do not differ from regular breakpoints. You can set conditions or commands, enable and disable them and perform other breakpoint operations.

GDB provides some additional commands for controlling what happens when the '`break`' command cannot resolve breakpoint address specification to an address:

`set breakpoint pending auto`
: This is the default behavior. When GDB cannot find the breakpoint location, it queries you whether a pending breakpoint should be created.

`set breakpoint pending on`
: This indicates that an unrecognized breakpoint location should automatically result in a pending breakpoint being created.

`set breakpoint pending off`
: This indicates that pending breakpoints are not to be created. Any unrecognized breakpoint location results in an error. This setting does not affect any pending breakpoints previously created.

`show breakpoint pending`
: Show the current behavior setting for creating pending breakpoints.

The settings above only affect the `break` command and its variants. Once breakpoint is set, it will be automatically updated as shared libraries are loaded and unloaded.

For some targets, GDB can automatically decide if hardware or software breakpoints should be used, depending on whether the breakpoint address is read-only or read-write. This applies to breakpoints set with the `break` command as well as to internal breakpoints set by commands like `next` and `finish`. For breakpoints set with `hbreak`, GDB will always use hardware breakpoints.

You can control this automatic behaviour with the following commands:

`set breakpoint auto-hw on`
: This is the default behavior. When GDB sets a breakpoint, it will try to use the target memory map to decide if software or hardware breakpoint must be used.

`set breakpoint auto-hw off`
: This indicates GDB should not automatically select breakpoint type. If the target provides a memory map, GDB will warn when trying to set software breakpoint at a read-only address.

GDB normally implements breakpoints by replacing the program code at the breakpoint address with a special instruction, which, when executed, given control to the debugger. By default, the program code is so modified only when the program is resumed. As soon as the program stops, GDB restores the original instructions. This behaviour guards against leaving breakpoints inserted in the target should gdb abrubptly disconnect. However, with

slow remote targets, inserting and removing breakpoint can reduce the performance. This behavior can be controlled with the following commands::

`set breakpoint always-inserted off`
: All breakpoints, including newly added by the user, are inserted in the target only when the target is resumed. All breakpoints are removed from the target when it stops. This is the default mode.

`set breakpoint always-inserted on`
: Causes all breakpoints to be inserted in the target at all times. If the user adds a new breakpoint, or changes an existing breakpoint, the breakpoints in the target are updated immediately. A breakpoint is removed from the target only when breakpoint itself is deleted.

GDB handles conditional breakpoints by evaluating these conditions when a breakpoint breaks. If the condition is true, then the process being debugged stops, otherwise the process is resumed.

If the target supports evaluating conditions on its end, GDB may download the breakpoint, together with its conditions, to it.

This feature can be controlled via the following commands:

`set breakpoint condition-evaluation host`
: This option commands GDB to evaluate the breakpoint conditions on the host's side. Unconditional breakpoints are sent to the target which in turn receives the triggers and reports them back to GDB for condition evaluation. This is the standard evaluation mode.

`set breakpoint condition-evaluation target`
: This option commands GDB to download breakpoint conditions to the target at the moment of their insertion. The target is responsible for evaluating the conditional expression and reporting breakpoint stop events back to GDB whenever the condition is true. Due to limitations of target-side evaluation, some conditions cannot be evaluated there, e.g., conditions that depend on local data that is only known to the host. Examples include conditional expressions involving convenience variables, complex types that cannot be handled by the agent expression parser and expressions that are too long to be sent over to the target, specially when the target is a remote system. In these cases, the conditions will be evaluated by GDB.

`set breakpoint condition-evaluation auto`
: This is the default mode. If the target supports evaluating breakpoint conditions on its end, GDB will download breakpoint conditions to the target (limitations mentioned previously apply). If the target does not support breakpoint condition evaluation, then GDB will fallback to evaluating all these conditions on the host's side.

GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of `longjmp` (in C programs). These internal breakpoints are assigned negative numbers, starting with `-1`; '`info breakpoints`' does not display them. You can see these breakpoints with the GDB maintenance command '`maint info breakpoints`' (see [maint info breakpoints], page 671).

5.1.2 Setting Watchpoints

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen. (This is sometimes called a *data breakpoint*.) The expression may be as simple as the value of a single variable, or as complex as many variables combined by operators. Examples include:

- A reference to the value of a single variable.
- An address cast to an appropriate data type. For example, '`*(int *)0x12345678`' will watch a 4-byte region at the specified address (assuming an `int` occupies 4 bytes).
- An arbitrarily complex expression, such as '`a*b + c/d`'. The expression can use any operators valid in the program's native language (see Chapter 15 [Languages], page 217).

You can set a watchpoint on an expression even if the expression can not be evaluated yet. For instance, you can set a watchpoint on '`*global_ptr`' before '`global_ptr`' is initialized. GDB will stop when your program sets '`global_ptr`' and the expression produces a valid value. If the expression becomes valid in some other way than changing a variable (e.g. if the memory pointed to by '`*global_ptr`' becomes readable as the result of a `malloc` call), GDB may not stop until the next time the expression changes.

Depending on your system, watchpoints may be implemented in software or hardware. GDB does software watchpointing by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution. (But this may still be worth it, to catch errors where you have no clue what part of your program is the culprit.)

On some systems, such as most PowerPC or x86-based targets, GDB includes support for hardware watchpoints, which do not slow down the running of your program.

`watch` [`-l`|`-location`] *expr* [`thread` *thread-id*] [`mask` *maskvalue*] [`task` *task-id*]

Set a watchpoint for an expression. GDB will break when the expression *expr* is written into by the program and its value changes. The simplest (and the most popular) use of this command is to watch the value of a single variable:

```
(gdb) watch foo
```

If the command includes a [`thread` *thread-id*] argument, GDB breaks only when the thread identified by *thread-id* changes the value of *expr*. If any other threads change the value of *expr*, GDB will not break. Note that watchpoints restricted to a single thread in this way only work with Hardware Watchpoints.

Similarly, if the `task` argument is given, then the watchpoint will be specific to the indicated Ada task (see Section 15.4.10.7 [Ada Tasks], page 243).

Ordinarily a watchpoint respects the scope of variables in *expr* (see below). The `-location` argument tells GDB to instead watch the memory referred to by *expr*. In this case, GDB will evaluate *expr*, take the address of the result, and watch the memory at that address. The type of the result is used to determine the size of the watched memory. If the expression's result does not have an address, then GDB will print an error.

The [`mask` *maskvalue*] argument allows creation of masked watchpoints, if the current architecture supports this feature (e.g., PowerPC Embedded architecture, see Section 21.3.8 [PowerPC Embedded], page 336.) A *masked watchpoint* specifies a mask in addition to an address to watch. The mask specifies that

some bits of an address (the bits which are reset in the mask) should be ignored when matching the address accessed by the inferior against the watchpoint address. Thus, a masked watchpoint watches many addresses simultaneously—those addresses whose unmasked bits are identical to the unmasked bits in the watchpoint address. The `mask` argument implies `-location`. Examples:

```
(gdb) watch foo mask 0xffff00ff
(gdb) watch *0xdeadbeef mask 0xffffff00
```

`rwatch` [`-l`|`-location`] *expr* [`thread` *thread-id*] [`mask` *maskvalue*]
Set a watchpoint that will break when the value of *expr* is read by the program.

`awatch` [`-l`|`-location`] *expr* [`thread` *thread-id*] [`mask` *maskvalue*]
Set a watchpoint that will break when *expr* is either read from or written into by the program.

`info watchpoints` [*list...*]
This command prints a list of watchpoints, using the same format as `info break` (see Section 5.1.1 [Set Breaks], page 56).

If you watch for a change in a numerically entered address you need to dereference it, as the address itself is just a constant number which will never change. GDB refuses to create a watchpoint that watches a never-changing value:

```
(gdb) watch 0x600850
Cannot watch constant value 0x600850.
(gdb) watch *(int *) 0x600850
Watchpoint 1: *(int *) 6293584
```

GDB sets a *hardware watchpoint* if possible. Hardware watchpoints execute very quickly, and the debugger reports a change in value at the exact instruction where the change occurs. If GDB cannot set a hardware watchpoint, it sets a software watchpoint, which executes more slowly and reports the change in value at the next *statement*, not the instruction, after the change occurs.

You can force GDB to use only software watchpoints with the *`set can-use-hw-watchpoints 0`* command. With this variable set to zero, GDB will never try to use hardware watchpoints, even if the underlying system supports them. (Note that hardware-assisted watchpoints that were set *before* setting `can-use-hw-watchpoints` to zero will still use the hardware mechanism of watching expression values.)

`set can-use-hw-watchpoints`
Set whether or not to use hardware watchpoints.

`show can-use-hw-watchpoints`
Show the current mode of using hardware watchpoints.

For remote targets, you can restrict the number of hardware watchpoints GDB will use, see [set remote hardware-breakpoint-limit], page 310.

When you issue the `watch` command, GDB reports

```
Hardware watchpoint num: expr
```

if it was able to set a hardware watchpoint.

Currently, the `awatch` and `rwatch` commands can only set hardware watchpoints, because accesses to data that don't change the value of the watched expression cannot be

detected without examining every instruction as it is being executed, and GDB does not do that currently. If GDB finds that it is unable to set a hardware breakpoint with the `awatch` or `rwatch` command, it will print a message like this:

```
Expression cannot be implemented with read/access watchpoint.
```

Sometimes, GDB cannot set a hardware watchpoint because the data type of the watched expression is wider than what a hardware watchpoint on the target machine can handle. For example, some systems can only watch regions that are up to 4 bytes wide; on such systems you cannot set hardware watchpoints for an expression that yields a double-precision floating-point number (which is typically 8 bytes wide). As a work-around, it might be possible to break the large region into a series of smaller ones and watch them with separate watchpoints.

If you set too many hardware watchpoints, GDB might be unable to insert all of them when you resume the execution of your program. Since the precise number of active watchpoints is unknown until such time as the program is about to be resumed, GDB might not be able to warn you about this when you set the watchpoints, and the warning will be printed only when the program is resumed:

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

If this happens, delete or disable some of the watchpoints.

Watching complex expressions that reference many variables can also exhaust the resources available for hardware-assisted watchpoints. That's because GDB needs to watch every variable in the expression with separately allocated resources.

If you call a function interactively using `print` or `call`, any watchpoints you have set will be inactive until GDB reaches another kind of breakpoint or the call completes.

GDB automatically deletes watchpoints that watch local (automatic) variables, or expressions that involve such variables, when they go out of scope, that is, when the execution leaves the block in which these variables were defined. In particular, when the program being debugged terminates, *all* local variables go out of scope, and so only watchpoints that watch global variables remain set. If you rerun the program, you will need to set all such watchpoints again. One way of doing that would be to set a code breakpoint at the entry to the `main` function and when it breaks, set all the watchpoints.

In multi-threaded programs, watchpoints will detect changes to the watched expression from every thread.

> *Warning:* In multi-threaded programs, software watchpoints have only limited usefulness. If GDB creates a software watchpoint, it can only watch the value of an expression *in a single thread*. If you are confident that the expression can only change due to the current thread's activity (and if you are also confident that no other thread can become current), then you can use software watchpoints as usual. However, GDB may not notice when a non-current thread's activity changes the expression. (Hardware watchpoints, in contrast, watch an expression in all threads.)

See [set remote hardware-watchpoint-limit], page 310.

5.1.3 Setting Catchpoints

You can use *catchpoints* to cause the debugger to stop for certain kinds of program events, such as C++ exceptions or the loading of a shared library. Use the `catch` command to set a catchpoint.

`catch` *event*

Stop when *event* occurs. The *event* can be any of the following:

`throw` [*regexp*]
`rethrow` [*regexp*]
`catch` [*regexp*]

The throwing, re-throwing, or catching of a C++ exception.

If *regexp* is given, then only exceptions whose type matches the regular expression will be caught.

The convenience variable `$_exception` is available at an exception-related catchpoint, on some systems. This holds the exception being thrown.

There are currently some limitations to C++ exception handling in GDB:

- The support for these commands is system-dependent. Currently, only systems using the '`gnu-v3`' C++ ABI (see Section 22.7 [ABI], page 352) are supported.
- The regular expression feature and the `$_exception` convenience variable rely on the presence of some SDT probes in `libstdc++`. If these probes are not present, then these features cannot be used. These probes were first available in the GCC 4.8 release, but whether or not they are available in your GCC also depends on how it was built.
- The `$_exception` convenience variable is only valid at the instruction at which an exception-related catchpoint is set.
- When an exception-related catchpoint is hit, GDB stops at a location in the system library which implements runtime exception support for C++, usually `libstdc++`. You can use `up` (see Section 8.3 [Selection], page 111) to get to your code.
- If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program either to abort or to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits. This is the case even if you set a catchpoint for the exception; catchpoints on exceptions are disabled within interactive calls. See Section 17.5 [Calling], page 269, for information on controlling this with `set unwind-on-terminating-exception`.
- You cannot raise an exception interactively.
- You cannot install an exception handler interactively.

`exception` [*name*]
: An Ada exception being raised. If an exception name is specified at the end of the command (eg `catch exception Program_Error`), the debugger will stop only when this specific exception is raised. Otherwise, the debugger stops execution when any Ada exception is raised.

  When inserting an exception catchpoint on a user-defined exception whose name is identical to one of the exceptions defined by the language, the fully qualified name must be used as the exception name. Otherwise, GDB will assume that it should stop on the pre-defined exception rather than the user-defined one. For instance, assuming an exception called `Constraint_Error` is defined in package `Pck`, then the command to use to catch such exceptions is *`catch exception Pck.Constraint_Error`*.

  The convenience variable `$_ada_exception` holds the address of the exception being thrown. This can be useful when setting a condition for such a catchpoint.

`exception unhandled`
: An exception that was raised but is not handled by the program. The convenience variable `$_ada_exception` is set as for `catch exception`.

`handlers` [*name*]
: An Ada exception being handled. If an exception name is specified at the end of the command (eg *`catch handlers Program_Error`*), the debugger will stop only when this specific exception is handled. Otherwise, the debugger stops execution when any Ada exception is handled.

  When inserting a handlers catchpoint on a user-defined exception whose name is identical to one of the exceptions defined by the language, the fully qualified name must be used as the exception name. Otherwise, GDB will assume that it should stop on the pre-defined exception rather than the user-defined one. For instance, assuming an exception called `Constraint_Error` is defined in package `Pck`, then the command to use to catch such exceptions handling is *`catch handlers Pck.Constraint_Error`*.

  The convenience variable `$_ada_exception` is set as for `catch exception`.

`assert`
: A failed Ada assertion. Note that the convenience variable `$_ada_exception` is *not* set by this catchpoint.

`exec`
: A call to `exec`.

`syscall`
`syscall` [*name* | *number* | group:*groupname* | g:*groupname*] ...
: A call to or return from a system call, a.k.a. *syscall*. A syscall is a mechanism for application programs to request a service from the

operating system (OS) or one of the OS system services. GDB can catch some or all of the syscalls issued by the debuggee, and show the related information for each syscall. If no argument is specified, calls to and returns from all system calls will be caught.

name can be any system call name that is valid for the underlying OS. Just what syscalls are valid depends on the OS. On GNU and Unix systems, you can find the full list of valid syscall names on `/usr/include/asm/unistd.h`.

Normally, GDB knows in advance which syscalls are valid for each OS, so you can use the GDB command-line completion facilities (see Section 3.3 [command completion], page 24) to list the available choices.

You may also specify the system call numerically. A syscall's number is the value passed to the OS's syscall dispatcher to identify the requested service. When you specify the syscall by its name, GDB uses its database of syscalls to convert the name into the corresponding numeric code, but using the number directly may be useful if GDB's database does not have the complete list of syscalls on your system (e.g., because GDB lags behind the OS upgrades).

You may specify a group of related syscalls to be caught at once using the `group:` syntax (`g:` is a shorter equivalent). For instance, on some platforms GDB allows you to catch all network related syscalls, by passing the argument `group:network` to `catch syscall`. Note that not all syscall groups are available in every system. You can use the command completion facilities (see Section 3.3 [command completion], page 24) to list the syscall groups available on your environment.

The example below illustrates how this command works if you don't provide arguments to it:

```
(gdb) catch syscall
Catchpoint 1 (syscall)
(gdb) r
Starting program: /tmp/catch-syscall

Catchpoint 1 (call to syscall 'close'), \
   0xffffe424 in __kernel_vsyscall ()
(gdb) c
Continuing.

Catchpoint 1 (returned from syscall 'close'), \
0xffffe424 in __kernel_vsyscall ()
(gdb)
```

Here is an example of catching a system call by name:

```
(gdb) catch syscall chroot
Catchpoint 1 (syscall 'chroot' [61])
(gdb) r
Starting program: /tmp/catch-syscall

Catchpoint 1 (call to syscall 'chroot'), \
```

```
   0xffffe424 in __kernel_vsyscall ()
(gdb) c
Continuing.

Catchpoint 1 (returned from syscall 'chroot'), \
0xffffe424 in __kernel_vsyscall ()
(gdb)
```

An example of specifying a system call numerically. In the case below, the syscall number has a corresponding entry in the XML file, so GDB finds its name and prints it:

```
(gdb) catch syscall 252
Catchpoint 1 (syscall(s) 'exit_group')
(gdb) r
Starting program: /tmp/catch-syscall

Catchpoint 1 (call to syscall 'exit_group'), \
   0xffffe424 in __kernel_vsyscall ()
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

Here is an example of catching a syscall group:

```
(gdb) catch syscall group:process
Catchpoint 1 (syscalls 'exit' [1] 'fork' [2] 'waitpid' [7]
'execve' [11] 'wait4' [114] 'clone' [120] 'vfork' [190]
'exit_group' [252] 'waitid' [284] 'unshare' [310])
(gdb) r
Starting program: /tmp/catch-syscall

Catchpoint 1 (call to syscall fork), 0x00007ffff7df4e27 in open64 ()
   from /lib64/ld-linux-x86-64.so.2

(gdb) c
Continuing.
```

However, there can be situations when there is no corresponding name in XML file for that syscall number. In this case, GDB prints a warning message saying that it was not able to find the syscall name, but the catchpoint will be set anyway. See the example below:

```
(gdb) catch syscall 764
warning: The number '764' does not represent a known syscall.
Catchpoint 2 (syscall 764)
(gdb)
```

If you configure GDB using the '`--without-expat`' option, it will not be able to display syscall names. Also, if your architecture does not have an XML file describing its system calls, you will not be able to see the syscall names. It is important to notice that these two features are used for accessing the syscall name database. In either case, you will see a warning like this:

```
(gdb) catch syscall
warning: Could not open "syscalls/i386-linux.xml"
warning: Could not load the syscall XML file 'syscalls/i386-linux.xml'.
```

```
GDB will not be able to display syscall names.
Catchpoint 1 (syscall)
(gdb)
```

Of course, the file name will change depending on your architecture and system.

Still using the example above, you can also try to catch a syscall by its number. In this case, you would see something like:

```
(gdb) catch syscall 252
Catchpoint 1 (syscall(s) 252)
```

Again, in this case GDB would not be able to display syscall's names.

`fork` A call to `fork`.

`vfork` A call to `vfork`.

`load` [*regexp*]
`unload` [*regexp*]
The loading or unloading of a shared library. If *regexp* is given, then the catchpoint will stop only if the regular expression matches one of the affected libraries.

`signal` [*signal...* | `'all'`]
The delivery of a signal.

With no arguments, this catchpoint will catch any signal that is not used internally by GDB, specifically, all signals except '`SIGTRAP`' and '`SIGINT`'.

With the argument '`all`', all signals, including those used by GDB, will be caught. This argument cannot be used with other signal names.

Otherwise, the arguments are a list of signal names as given to `handle` (see Section 5.4 [Signals], page 86). Only signals specified in this list will be caught.

One reason that `catch signal` can be more useful than `handle` is that you can attach commands and conditions to the catchpoint.

When a signal is caught by a catchpoint, the signal's `stop` and `print` settings, as specified by `handle`, are ignored. However, whether the signal is still delivered to the inferior depends on the `pass` setting; this can be changed in the catchpoint's commands.

`tcatch` *event*
Set a catchpoint that is enabled only for one stop. The catchpoint is automatically deleted after the first time the event is caught.

Use the `info break` command to list the current catchpoints.

5.1.4 Deleting Breakpoints

It is often necessary to eliminate a breakpoint, watchpoint, or catchpoint once it has done its job and you no longer want your program to stop there. This is called *deleting* the breakpoint. A breakpoint that has been deleted no longer exists; it is forgotten.

With the `clear` command you can delete breakpoints according to where they are in your program. With the `delete` command you can delete individual breakpoints, watchpoints, or catchpoints by specifying their breakpoint numbers.

It is not necessary to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints on the first instruction to be executed when you continue execution without changing the execution address.

`clear` Delete any breakpoints at the next instruction to be executed in the selected stack frame (see Section 8.3 [Selecting a Frame], page 111). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

`clear` *location*
Delete any breakpoints set at the specified *location*. See Section 9.2 [Specify Location], page 120, for the various forms of *location*; the most useful ones are listed below:

`clear` *function*
`clear` *filename*`:`*function*
Delete any breakpoints set at entry to the named *function*.

`clear` *linenum*
`clear` *filename*`:`*linenum*
Delete any breakpoints set at or within the code of the specified *linenum* of the specified *filename*.

`delete` [`breakpoints`] [*list*`...`]
Delete the breakpoints, watchpoints, or catchpoints of the breakpoint list specified as argument. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have `set confirm off`). You can abbreviate this command as `d`.

5.1.5 Disabling Breakpoints

Rather than deleting a breakpoint, watchpoint, or catchpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can *enable* it again later.

You disable and enable breakpoints, watchpoints, and catchpoints with the `enable` and `disable` commands, optionally specifying one or more breakpoint numbers as arguments. Use `info break` to print a list of all breakpoints, watchpoints, and catchpoints if you do not know which numbers to use.

Disabling and enabling a breakpoint that has multiple locations affects all of its locations.

A breakpoint, watchpoint, or catchpoint can have any of several different states of enablement:

- Enabled. The breakpoint stops your program. A breakpoint set with the `break` command starts out in this state.
- Disabled. The breakpoint has no effect on your program.
- Enabled once. The breakpoint stops your program, but then becomes disabled.

- Enabled for a count. The breakpoint stops your program for the next N times, then becomes disabled.
- Enabled for deletion. The breakpoint stops your program, but immediately after it does so it is deleted permanently. A breakpoint set with the `tbreak` command starts out in this state.

You can use the following commands to enable or disable breakpoints, watchpoints, and catchpoints:

`disable` [`breakpoints`] [*`list...`*]
: Disable the specified breakpoints—or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later. You may abbreviate `disable` as `dis`.

`enable` [`breakpoints`] [*`list...`*]
: Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

`enable` [`breakpoints`] `once` *`list...`*
: Enable the specified breakpoints temporarily. GDB disables any of these breakpoints immediately after stopping your program.

`enable` [`breakpoints`] `count` *`count`* *`list...`*
: Enable the specified breakpoints temporarily. GDB records *count* with each of the specified breakpoints, and decrements a breakpoint's count when it is hit. When any count reaches 0, GDB disables that breakpoint. If a breakpoint has an ignore count (see Section 5.1.6 [Break Conditions], page 72), that will be decremented to 0 before *count* is affected.

`enable` [`breakpoints`] `delete` *`list...`*
: Enable the specified breakpoints to work once, then die. GDB deletes any of these breakpoints as soon as your program stops there. Breakpoints set by the `tbreak` command start out in this state.

Except for a breakpoint set with `tbreak` (see Section 5.1.1 [Setting Breakpoints], page 56), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see Section 5.2 [Continuing and Stepping], page 80.)

5.1.6 Break Conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your programming language (see Section 10.1 [Expressions], page 137). A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated—that is, when the condition is false. In C, if you want to test an assertion expressed by the condition *assert*, you should set the condition '`! assert`' on the appropriate breakpoint.

Conditions are also accepted for watchpoints; you may not need them, since a watchpoint is inspecting the value of an expression anyhow—but it might be simpler, say, to just set a watchpoint on a variable name, and specify a condition that tests whether the new value is an interesting one.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible than break conditions for the purpose of performing side effects when a breakpoint is reached (see Section 5.1.7 [Breakpoint Command Lists], page 74).

Breakpoint conditions can also be evaluated on the target's side if the target supports it. Instead of evaluating the conditions locally, GDB encodes the expression into an agent expression (see Appendix F [Agent Expressions], page 757) suitable for execution on the target, independently of GDB. Global variables become raw memory locations, locals become stack accesses, and so forth.

In this case, GDB will only be notified of a breakpoint trigger when its condition evaluates to true. This mechanism may provide faster response times depending on the performance characteristics of the target since it does not need to keep GDB informed about every breakpoint trigger, even those with false conditions.

Break conditions can be specified when a breakpoint is set, by using '`if`' in the arguments to the `break` command. See Section 5.1.1 [Setting Breakpoints], page 56. They can also be changed at any time with the `condition` command.

You can also use the `if` keyword with the `watch` command. The `catch` command does not recognize the `if` keyword; `condition` is the only way to impose a further condition on a catchpoint.

`condition` *bnum expression*
: Specify *expression* as the break condition for breakpoint, watchpoint, or catchpoint number *bnum*. After you set a condition, breakpoint *bnum* stops your program only if the value of *expression* is true (nonzero, in C). When you use `condition`, GDB checks *expression* immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. If *expression* uses symbols not referenced in the context of the breakpoint, GDB prints an error message:

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

GDB does not actually evaluate *expression* at the time the `condition` command (or a command that sets a breakpoint with a condition, like `break if ...`) is given, however. See Section 10.1 [Expressions], page 137.

`condition -force` *bnum expression*
: When the `-force` flag is used, define the condition even if *expression* is invalid at all the current locations of breakpoint *bnum*. This is similar to the `-force-condition` option of the `break` command.

`condition` *bnum*
: Remove the condition from breakpoint number *bnum*. It becomes an ordinary unconditional breakpoint.

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. This is so useful that there is a special way to do it, using the *ignore count* of the breakpoint. Every breakpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if your program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is *n*, the breakpoint does not stop the next *n* times your program reaches it.

`ignore` *bnum count*
: Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, your program's execution does not stop; other than to decrement the ignore count, GDB takes no action.

  To make the breakpoint stop the next time it is reached, specify a count of zero.

  When you use `continue` to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to `continue`, rather than using `ignore`. See Section 5.2 [Continuing and Stepping], page 80.

  If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, GDB resumes checking the condition.

  You could achieve the effect of the ignore count with a condition such as '`$foo-- <= 0`' using a debugger convenience variable that is decremented each time. See Section 10.12 [Convenience Variables], page 162.

Ignore counts apply to breakpoints, watchpoints, and catchpoints.

5.1.7 Breakpoint Command Lists

You can give any breakpoint (or watchpoint or catchpoint) a series of commands to execute when your program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

`commands` [*list...*]
... command-list ...
`end`
: Specify a list of commands for the given breakpoints. The commands themselves appear on the following lines. Type a line containing just `end` to terminate the commands.

  To remove all commands from a breakpoint, type `commands` and follow it immediately with `end`; that is, give no commands.

  With no argument, `commands` refers to the last breakpoint, watchpoint, or catchpoint set (not to the breakpoint most recently encountered). If the most recent breakpoints were set with a single command, then the `commands` will apply to all the breakpoints set by that command. This applies to breakpoints set by `rbreak`, and also applies when a single `break` command creates multiple breakpoints (see Section 10.2 [Ambiguous Expressions], page 137).

Pressing `RET` as a means of repeating the last GDB command is disabled within a *command-list*.

You can use breakpoint commands to start your program up again. Simply use the `continue` command, or `step`, or any other command that resumes execution.

Any other commands in the command list, after a command that resumes execution, are ignored. This is because any time you resume execution (even with a simple `next` or `step`), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If the first command you specify in a command list is `silent`, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the remaining commands print anything, you see no sign that the breakpoint was reached. `silent` is meaningful only at the beginning of a breakpoint command list.

The commands `echo`, `output`, and `printf` allow you to print precisely controlled output, and are often useful in silent breakpoints. See Section 23.1.4 [Commands for Controlled Output], page 372.

For example, here is how you could use breakpoint commands to print the value of `x` at entry to `foo` whenever `x` is positive.

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

One application for breakpoint commands is to compensate for one bug so you can test for another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the `continue` command so that your program does not stop, and start with the `silent` command so that no output is produced. Here is an example:

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

5.1.8 Dynamic Printf

The dynamic printf command `dprintf` combines a breakpoint with formatted printing of your program's data to give you the effect of inserting `printf` calls into your program on-the-fly, without having to recompile it.

In its most basic form, the output goes to the GDB console. However, you can set the variable `dprintf-style` for alternate handling. For instance, you can ask to format the output by calling your program's `printf` function. This has the advantage that the characters go to the program's output device, so they can recorded in redirects to files and so forth.

If you are doing remote debugging with a stub or agent, you can also ask to have the printf handled by the remote agent. In addition to ensuring that the output goes to the remote program's device along with any other output the program might produce, you can also ask that the dprintf remain active even after disconnecting from the remote target. Using the stub/agent is also more efficient, as it can do everything without needing to communicate with GDB.

`dprintf` *`location`*`,`*`template`*`,`*`expression`*`[,`*`expression`*`...]`
: Whenever execution reaches *location*, print the values of one or more *expressions* under the control of the string *template*. To print several values, separate them with commas.

`set dprintf-style` *`style`*
: Set the dprintf output to be handled in one of several different styles enumerated below. A change of style affects all existing dynamic printfs immediately. (If you need individual control over the print commands, simply define normal breakpoints with explicitly-supplied command lists.)

  `gdb` — Handle the output using the GDB `printf` command.

  `call` — Handle the output by calling a function in your program (normally `printf`).

  `agent` — Have the remote debugging agent (such as `gdbserver`) handle the output itself. This style is only available for agents that support running commands on the target.

`set dprintf-function` *`function`*
: Set the function to call if the dprintf style is `call`. By default its value is `printf`. You may set it to any expression. that GDB can evaluate to a function, as per the `call` command.

`set dprintf-channel` *`channel`*
: Set a "channel" for dprintf. If set to a non-empty value, GDB will evaluate it as an expression and pass the result as a first argument to the `dprintf-function`, in the manner of `fprintf` and similar functions. Otherwise, the dprintf format string will be the first argument, in the manner of `printf`.

  As an example, if you wanted `dprintf` output to go to a logfile that is a standard I/O stream assigned to the variable `mylog`, you could do the following:

```
(gdb) set dprintf-style call
(gdb) set dprintf-function fprintf
(gdb) set dprintf-channel mylog
(gdb) dprintf 25,"at line 25, glob=%d\n",glob
Dprintf 1 at 0x123456: file main.c, line 25.
(gdb) info break
1       dprintf        keep y   0x00123456 in main at main.c:25
        call (void) fprintf (mylog,"at line 25, glob=%d\n",glob)
        continue
(gdb)
```

  Note that the `info break` displays the dynamic printf commands as normal breakpoint commands; you can thus easily see the effect of the variable settings.

`set disconnected-dprintf on`
`set disconnected-dprintf off`
: Choose whether `dprintf` commands should continue to run if GDB has disconnected from the target. This only applies if the `dprintf-style` is `agent`.

`show disconnected-dprintf off`
: Show the current choice for disconnected `dprintf`.

GDB does not check the validity of function and channel, relying on you to supply values that are meaningful for the contexts in which they are being used. For instance, the function and channel may be the values of local variables, but if that is the case, then all enabled dynamic prints must be at locations within the scope of those locals. If evaluation fails, GDB will report an error.

5.1.9 How to save breakpoints to a file

To save breakpoint definitions to a file use the `save breakpoints` command.

`save breakpoints [`*`filename`*`]`
: This command saves all current breakpoint definitions together with their commands and ignore counts, into a file *`filename`* suitable for use in a later debugging session. This includes all types of breakpoints (breakpoints, watchpoints, catchpoints, tracepoints). To read the saved breakpoint definitions, use the `source` command (see Section 23.1.3 [Command Files], page 371). Note that watchpoints with expressions involving local variables may fail to be recreated because it may not be possible to access the context where the watchpoint is valid anymore. Because the saved breakpoint definitions are simply a sequence of GDB commands that recreate the breakpoints, you can edit the file in your favorite editing program, and remove the breakpoint definitions you're not interested in, or that can no longer be recreated.

5.1.10 Static Probe Points

GDB supports *SDT* probes in the code. SDT stands for Statically Defined Tracing, and the probes are designed to have a tiny runtime code and data footprint, and no dynamic relocations.

Currently, the following types of probes are supported on ELF-compatible systems:

- `SystemTap` (`http://sourceware.org/systemtap/`) SDT probes[1]. `SystemTap` probes are usable from assembly, C and C++ languages[2].
- `DTrace` (`http://oss.oracle.com/projects/DTrace`) USDT probes. `DTrace` probes are usable from C and C++ languages.

Some `SystemTap` probes have an associated semaphore variable; for instance, this happens automatically if you defined your probe using a DTrace-style `.d` file. If your probe has a semaphore, GDB will automatically enable it when you specify a breakpoint using the '`-probe-stap`' notation. But, if you put a breakpoint at a probe's location by some

[1] See `http://sourceware.org/systemtap/wiki/AddingUserSpaceProbingToApps` for more information on how to add `SystemTap` SDT probes in your applications.

[2] See `http://sourceware.org/systemtap/wiki/UserSpaceProbeImplementation` for a good reference on how the SDT probes are implemented.

other method (e.g., `break file:line`), then GDB will not automatically set the semaphore. `DTrace` probes do not support semaphores.

You can examine the available static static probes using `info probes`, with optional arguments:

`info probes` [*type*] [*provider* [*name* [*objfile*]]]
: If given, *type* is either `stap` for listing `SystemTap` probes or `dtrace` for listing `DTrace` probes. If omitted all probes are listed regardless of their types.

  If given, *provider* is a regular expression used to match against provider names when selecting which probes to list. If omitted, probes by all probes from all providers are listed.

  If given, *name* is a regular expression to match against probe names when selecting which probes to list. If omitted, probe names are not considered when deciding whether to display them.

  If given, *objfile* is a regular expression used to select which object files (executable or shared libraries) to examine. If not given, all object files are considered.

`info probes all`
: List the available static probes, from all types.

Some probe points can be enabled and/or disabled. The effect of enabling or disabling a probe depends on the type of probe being handled. Some `DTrace` probes can be enabled or disabled, but `SystemTap` probes cannot be disabled.

You can enable (or disable) one or more probes using the following commands, with optional arguments:

`enable probes` [*provider* [*name* [*objfile*]]]
: If given, *provider* is a regular expression used to match against provider names when selecting which probes to enable. If omitted, all probes from all providers are enabled.

  If given, *name* is a regular expression to match against probe names when selecting which probes to enable. If omitted, probe names are not considered when deciding whether to enable them.

  If given, *objfile* is a regular expression used to select which object files (executable or shared libraries) to examine. If not given, all object files are considered.

`disable probes` [*provider* [*name* [*objfile*]]]
: See the `enable probes` command above for a description of the optional arguments accepted by this command.

A probe may specify up to twelve arguments. These are available at the point at which the probe is defined—that is, when the current PC is at the probe's location. The arguments are available using the convenience variables (see Section 10.12 [Convenience Vars], page 162) `$_probe_arg0`...`$_probe_arg11`. In `SystemTap` probes each probe argument is an integer of the appropriate size; types are not preserved. In `DTrace` probes types are preserved provided that they are recognized as such by GDB; otherwise the value of the

probe argument will be a long integer. The convenience variable `$_probe_argc` holds the number of arguments at the current probe point.

These variables are always available, but attempts to access them at any location other than a probe point will cause GDB to give an error message.

5.1.11 "Cannot insert breakpoints"

If you request too many active hardware-assisted breakpoints and watchpoints, you will see this error message:

```
Stopped; cannot insert breakpoints.
You may have requested too many hardware breakpoints and watchpoints.
```

This message is printed when you attempt to resume the program, since only then GDB knows exactly how many hardware breakpoints and watchpoints it needs to insert.

When this message is printed, you need to disable or remove some of the hardware-assisted breakpoints and watchpoints, and then continue.

5.1.12 "Breakpoint address adjusted..."

Some processor architectures place constraints on the addresses at which breakpoints may be placed. For architectures thus constrained, GDB will attempt to adjust the breakpoint's address to comply with the constraints dictated by the architecture.

One example of such an architecture is the Fujitsu FR-V. The FR-V is a VLIW architecture in which a number of RISC-like instructions may be bundled together for parallel execution. The FR-V architecture constrains the location of a breakpoint instruction within such a bundle to the instruction with the lowest address. GDB honors this constraint by adjusting a breakpoint's address to the first in the bundle.

It is not uncommon for optimized code to have bundles which contain instructions from different source statements, thus it may happen that a breakpoint's address will be adjusted from one source statement to another. Since this adjustment may significantly alter GDB's breakpoint related behavior from what the user expects, a warning is printed when the breakpoint is first set and also when the breakpoint is hit.

A warning like the one below is printed when setting a breakpoint that's been subject to address adjustment:

```
warning: Breakpoint address adjusted from 0x00010414 to 0x00010410.
```

Such warnings are printed both for user settable and GDB's internal breakpoints. If you see one of these warnings, you should verify that a breakpoint set at the adjusted address will have the desired affect. If not, the breakpoint in question may be removed and other breakpoints may be set which will have the desired behavior. E.g., it may be sufficient to place the breakpoint at a later instruction. A conditional breakpoint may also be useful in some cases to prevent the breakpoint from triggering too often.

GDB will also issue a warning when stopping at one of these adjusted breakpoints:

```
warning: Breakpoint 1 address previously adjusted from 0x00010414
to 0x00010410.
```

When this warning is encountered, it may be too late to take remedial action except in cases where the breakpoint is hit earlier or more frequently than expected.

5.2 Continuing and Stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If it stops due to a signal, you may want to use `handle`, or use '`signal 0`' to resume execution (see Section 5.4 [Signals], page 86), or you may step into the signal's handler (see [stepping and signal handlers], page 87).)

`continue` [*ignore-count*]
`c` [*ignore-count*]
`fg` [*ignore-count*]

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of `ignore` (see Section 5.1.6 [Break Conditions], page 72).

The argument *ignore-count* is meaningful only when your program stopped due to a breakpoint. At other times, the argument to `continue` is ignored.

The synonyms `c` and `fg` (for *foreground*, as the debugged program is deemed to be the foreground program) are provided purely for convenience, and have exactly the same behavior as `continue`.

To resume execution at a different place, you can use `return` (see Section 17.4 [Returning from a Function], page 268) to go back to the calling function; or `jump` (see Section 17.2 [Continuing at a Different Address], page 266) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see Section 5.1 [Breakpoints; Watchpoints; and Catchpoints], page 55) at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

`step` Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated `s`.

> *Warning:* If you use the `step` command while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, it will not step into a function which is compiled without debugging information. To step through functions without debugging information, use the `stepi` command, described below.

The `step` command only stops at the first instruction of a source line. This prevents the multiple stops that could otherwise occur in `switch` statements, `for` loops, etc. `step` continues to stop if a function that has debugging information is called within the line. In other words, `step` *steps inside* any functions called within the line.

Also, the `step` command only enters a function if there is line number information for the function. Otherwise it acts like the `next` command. This avoids problems when using `cc -gl` on MIPS machines. Previously, `step` entered subroutines if there was any debugging information about the routine.

`step` *count*
: Continue running as in `step`, but do so *count* times. If a breakpoint is reached, or a signal not related to stepping occurs before *count* steps, stepping stops right away.

`next` [*count*]
: Continue to the next source line in the current (innermost) stack frame. This is similar to `step`, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next` command. This command is abbreviated `n`.

  An argument *count* is a repeat count, as for `step`.

  The `next` command only stops at the first instruction of a source line. This prevents multiple stops that could otherwise occur in `switch` statements, `for` loops, etc.

`set step-mode`
`set step-mode on`
: The `set step-mode on` command causes the `step` command to stop at the first instruction of a function which contains no debug line information rather than stepping over it.

  This is useful in cases where you may be interested in inspecting the machine instructions of a function which has no symbolic info and do not want GDB to automatically skip over this function.

`set step-mode off`
: Causes the `step` command to step over any functions which contains no debug information. This is the default.

`show step-mode`
: Show whether GDB will stop in or step over functions without source line debug information.

`finish`
: Continue running until just after function in the selected stack frame returns. Print the returned value (if any). This command can be abbreviated as `fin`.

  Contrast this with the `return` command (see Section 17.4 [Returning from a Function], page 268).

`set print finish` [`on`|`off`]
`show print finish`
: By default the `finish` command will show the value that is returned by the function. This can be disabled using `set print finish off`. When disabled, the value is still entered into the value history (see Section 10.11 [Value History], page 161), but not displayed.

`until`
`u` Continue running until a source line past the current line, in the current stack frame, is reached. This command is used to avoid single stepping through a loop more than once. It is like the `next` command, except that when `until` encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.

This means that when you reach the end of a loop after single stepping though it, `until` makes your program continue execution until it exits the loop. In contrast, a `next` command at the end of a loop simply steps back to the beginning of the loop, which forces you to step through the next iteration.

`until` always stops your program if it attempts to exit the current stack frame.

`until` may produce somewhat counterintuitive results if the order of machine code does not match the order of the source lines. For example, in the following excerpt from a debugging session, the `f` (`frame`) command shows that execution is stopped at line `206`; yet when we use `until`, we get to line `195`:

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

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop—even though the test in a C `for`-loop is written before the body of the loop. The `until` command appeared to step back to the beginning of the loop when it advanced to this expression; however, it has not really gone to an earlier statement—not in terms of the actual machine code.

`until` with no argument works by means of single instruction stepping, and hence is slower than `until` with an argument.

`until` *location*
`u` *location*
Continue running your program until either the specified *location* is reached, or the current stack frame returns. The location is any of the forms described in Section 9.2 [Specify Location], page 120. This form of the command uses temporary breakpoints, and hence is quicker than `until` without an argument. The specified location is actually reached only if it is in the current frame. This implies that `until` can be used to skip over recursive function invocations. For instance in the code below, if the current location is line `96`, issuing `until 99` will execute the program up to line `99` in the same invocation of factorial, i.e., after the inner invocations have returned.

```
94 int factorial (int value)
95 {
96     if (value > 1) {
97             value *= factorial (value - 1);
98     }
99     return (value);
100     }
```

`advance` *location*
: Continue running the program up to the given *location*. An argument is required, which should be of one of the forms described in Section 9.2 [Specify Location], page 120. Execution will also stop upon exit from the current stack frame. This command is similar to `until`, but `advance` will not skip over recursive function calls, and the target location doesn't have to be in the same frame as the current one.

`stepi`
`stepi` *arg*
`si`
: Execute one machine instruction, then stop and return to the debugger.

  It is often useful to do '`display/i $pc`' when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See Section 10.8 [Automatic Display], page 147.

  An argument is a repeat count, as in `step`.

`nexti`
`nexti` *arg*
`ni`
: Execute one machine instruction, but if it is a function call, proceed until the function returns.

  An argument is a repeat count, as in `next`.

By default, and if available, GDB makes use of target-assisted *range stepping*. In other words, whenever you use a stepping command (e.g., `step`, `next`), GDB tells the target to step the corresponding range of instruction addresses instead of issuing multiple single-steps. This speeds up line stepping, particularly for remote targets. Ideally, there should be no reason you would want to turn range stepping off. However, it's possible that a bug in the debug info, a bug in the remote stub (for remote targets), or even a bug in GDB could make line stepping behave incorrectly when target-assisted range stepping is enabled. You can use the following command to turn off range stepping if necessary:

`set range-stepping`
`show range-stepping`
: Control whether range stepping is enabled.

  If `on`, and the target supports it, GDB tells the target to step a range of addresses itself, instead of issuing multiple single-steps. If `off`, GDB always issues single-steps, even if range stepping is supported by the target. The default is `on`.

5.3 Skipping Over Functions and Files

The program you are debugging may contain some functions which are uninteresting to debug. The `skip` command lets you tell GDB to skip a function, all functions in a file or a particular function in a particular file when stepping.

For example, consider the following C function:

```
101     int func()
102     {
103         foo(boring());
104         bar(boring());
105     }
```

Suppose you wish to step into the functions `foo` and `bar`, but you are not interested in stepping through `boring`. If you run `step` at line 103, you'll enter `boring()`, but if you run `next`, you'll step over both `foo` and `boring`!

One solution is to `step` into `boring` and use the `finish` command to immediately exit it. But this can become tedious if `boring` is called from many places.

A more flexible solution is to execute *`skip boring`*. This instructs GDB never to step into `boring`. Now when you execute `step` at line 103, you'll step over `boring` and directly into `foo`.

Functions may be skipped by providing either a function name, linespec (see Section 9.2 [Specify Location], page 120), regular expression that matches the function's name, file name or a `glob`-style pattern that matches the file name.

On Posix systems the form of the regular expression is "Extended Regular Expressions". See for example '`man 7 regex`' on GNU/Linux systems. On non-Posix systems the form of the regular expression is whatever is provided by the `regcomp` function of the underlying system. See for example '`man 7 glob`' on GNU/Linux systems for a description of `glob`-style patterns.

`skip` [*`options`*]

The basic form of the `skip` command takes zero or more options that specify what to skip. The *options* argument is any useful combination of the following:

`-file` *`file`*
`-fi` *`file`* Functions in *file* will be skipped over when stepping.

`-gfile` *`file-glob-pattern`*
`-gfi` *`file-glob-pattern`*
Functions in files matching *file-glob-pattern* will be skipped over when stepping.

```
(gdb) skip -gfi utils/*.c
```

`-function` *`linespec`*
`-fu` *`linespec`*
Functions named by *linespec* or the function containing the line named by *linespec* will be skipped over when stepping. See Section 9.2 [Specify Location], page 120.

`-rfunction` *`regexp`*
`-rfu` *`regexp`*
Functions whose name matches *regexp* will be skipped over when stepping.

This form is useful for complex function names. For example, there is generally no need to step into C++ `std::string` constructors or destructors. Plus with C++ templates it can be hard to write out the full name of the function, and often it doesn't matter what the template arguments are. Specifying the function to be skipped as a regular expression makes this easier.

```
(gdb) skip -rfu ^std::(allocator|basic_string)<.*>::~?\1 *\(
```

If you want to skip every templated C++ constructor and destructor in the `std` namespace you can do:

```
(gdb) skip -rfu ^std::([a-zA-z0-9_]+)<.*>::~?\1 *\(
```

If no options are specified, the function you're currently debugging will be skipped.

`skip function` [*linespec*]
: After running this command, the function named by *linespec* or the function containing the line named by *linespec* will be skipped over when stepping. See Section 9.2 [Specify Location], page 120.

  If you do not specify *linespec*, the function you're currently debugging will be skipped.

  (If you have a function called `file` that you want to skip, use *`skip function file`*.)

`skip file` [*filename*]
: After running this command, any function whose source lives in *filename* will be skipped over when stepping.

```
(gdb) skip file boring.c
File boring.c will be skipped when stepping.
```

  If you do not specify *filename*, functions whose source lives in the file you're currently debugging will be skipped.

Skips can be listed, deleted, disabled, and enabled, much like breakpoints. These are the commands for managing your list of skips:

`info skip` [*range*]
: Print details about the specified skip(s). If *range* is not specified, print a table with details about all functions and files marked for skipping. `info skip` prints the following information about each skip:

  Identifier A number identifying this skip.

  Enabled or Disabled
  : Enabled skips are marked with '`y`'. Disabled skips are marked with '`n`'.

  Glob If the file name is a '`glob`' pattern this is '`y`'. Otherwise it is '`n`'.

  File The name or '`glob`' pattern of the file to be skipped. If no file is specified this is '`<none>`'.

  RE If the function name is a '`regular expression`' this is '`y`'. Otherwise it is '`n`'.

  Function The name or regular expression of the function to skip. If no function is specified this is '`<none>`'.

`skip delete` [*range*]
: Delete the specified skip(s). If *range* is not specified, delete all skips.

`skip enable` [*range*]
: Enable the specified skip(s). If *range* is not specified, enable all skips.

`skip disable` [*range*]
: Disable the specified skip(s). If *range* is not specified, disable all skips.

`set debug skip` [`on|off`]
: Set whether to print the debug output about skipping files and functions.

`show debug skip`
: Show whether the debug output about skipping files and functions is printed.

5.4 Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, in Unix `SIGINT` is the signal a program gets when you type an interrupt character (often *Ctrl-c*); `SIGSEGV` is the signal a program gets from referencing a place in memory far away from all the areas in use; `SIGALRM` occurs when the alarm clock timer goes off (which happens only if your program has requested an alarm).

Some signals, including `SIGALRM`, are a normal part of the functioning of your program. Others, such as `SIGSEGV`, indicate errors; these signals are *fatal* (they kill your program immediately) if the program has not specified in advance some other way to handle the signal. `SIGINT` does not indicate an error in your program, but it is normally fatal so it can carry out the purpose of the interrupt: to kill the program.

GDB has the ability to detect any occurrence of a signal in your program. You can tell GDB in advance what to do for each kind of signal.

Normally, GDB is set up to let the non-erroneous signals like `SIGALRM` be silently passed to your program (so as not to interfere with their role in the program's functioning) but to stop your program immediately whenever an error signal happens. You can change these settings with the `handle` command.

`info signals`
`info handle`
: Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals.

`info signals` *sig*
: Similar, but print information only about the specified signal number.

  `info handle` is an alias for `info signals`.

`catch signal` [*signal...* | 'all']
: Set a catchpoint for the indicated signals. See Section 5.1.3 [Set Catchpoints], page 66, for details about this command.

`handle` *signal* [*keywords...*]
: Change the way GDB handles signal *signal*. The *signal* can be the number of a signal or its name (with or without the 'SIG' at the beginning); a list of signal numbers of the form '*low-high*'; or the word 'all', meaning all the known signals. Optional arguments *keywords*, described below, say what change to make.

The keywords allowed by the `handle` command can be abbreviated. Their full names are:

`nostop` GDB should not stop your program when this signal happens. It may still print a message telling you that the signal has come in.

`stop` GDB should stop your program when this signal happens. This implies the `print` keyword as well.

`print` GDB should print a message when this signal happens.

`noprint` GDB should not mention the occurrence of the signal at all. This implies the `nostop` keyword as well.

`pass`
`noignore` GDB should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled. `pass` and `noignore` are synonyms.

`nopass`
`ignore` GDB should not allow your program to see this signal. `nopass` and `ignore` are synonyms.

When a signal stops your program, the signal is not visible to the program until you continue. Your program sees the signal then, if `pass` is in effect for the signal in question *at that time*. In other words, after GDB reports a signal, you can use the `handle` command with `pass` or `nopass` to control whether your program sees that signal when you continue.

The default is set to `nostop`, `noprint`, `pass` for non-erroneous signals such as `SIGALRM`, `SIGWINCH` and `SIGCHLD`, and to `stop`, `print`, `pass` for the erroneous signals.

You can also use the `signal` command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with '`signal 0`'. See Section 17.3 [Giving your Program a Signal], page 267.

GDB optimizes for stepping the mainline code. If a signal that has `handle nostop` and `handle pass` set arrives while a stepping command (e.g., `stepi`, `step`, `next`) is in progress, GDB lets the signal handler run and then resumes stepping the mainline code once the signal handler returns. In other words, GDB steps over the signal handler. This prevents signals that you've specified as not interesting (with `handle nostop`) from changing the focus of debugging unexpectedly. Note that the signal handler itself may still hit a breakpoint, stop for another signal that has `handle stop` in effect, or for any other event that normally results in stopping the stepping command sooner. Also note that GDB still informs you that the program received a signal if `handle print` is set.

If you set `handle pass` for a signal, and your program sets up a handler for it, then issuing a stepping command, such as `step` or `stepi`, when your program is stopped due to the signal will step *into* the signal handler (if the target supports that).

Likewise, if you use the `queue-signal` command to queue a signal to be delivered to the current thread when execution of the thread resumes (see Section 17.3 [Giving your Program a Signal], page 267), then a stepping command will step into the signal handler.

Here's an example, using `stepi` to step to the first instruction of `SIGUSR1`'s handler:

```
(gdb) handle SIGUSR1
Signal        Stop      Print   Pass to program Description
SIGUSR1       Yes       Yes     Yes             User defined signal 1
(gdb) c
Continuing.

Program received signal SIGUSR1, User defined signal 1.
main () sigusr1.c:28
28        p = 0;
(gdb) si
sigusr1_handler () at sigusr1.c:9
9       {
```

The same, but using `queue-signal` instead of waiting for the program to receive the signal first:

```
(gdb) n
28        p = 0;
(gdb) queue-signal SIGUSR1
(gdb) si
sigusr1_handler () at sigusr1.c:9
9       {
(gdb)
```

On some targets, GDB can inspect extra signal information associated with the intercepted signal, before it is actually delivered to the program being debugged. This information is exported by the convenience variable `$_siginfo`, and consists of data that is passed by the kernel to the signal handler at the time of the receipt of a signal. The data type of the information itself is target dependent. You can see the data type using the `ptype $_siginfo` command. On Unix systems, it typically corresponds to the standard `siginfo_t` type, as defined in the `signal.h` system header.

Here's an example, on a GNU/Linux system, printing the stray referenced address that raised a segmentation fault.

```
(gdb) continue
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400766 in main ()
69        *(int *)p = 0;
(gdb) ptype $_siginfo
type = struct {
    int si_signo;
    int si_errno;
    int si_code;
    union {
        int _pad[28];
        struct {...} _kill;
        struct {...} _timer;
        struct {...} _rt;
        struct {...} _sigchld;
        struct {...} _sigfault;
        struct {...} _sigpoll;
    } _sifields;
}
(gdb) ptype $_siginfo._sifields._sigfault
type = struct {
    void *si_addr;
}
(gdb) p $_siginfo._sifields._sigfault.si_addr
$1 = (void *) 0x7ffff7ff7000
```

Depending on target support, `$_siginfo` may also be writable.

On some targets, a `SIGSEGV` can be caused by a boundary violation, i.e., accessing an address outside of the allowed range. In those cases GDB may displays additional information, depending on how GDB has been told to handle the signal. With `handle stop SIGSEGV`, GDB displays the violation kind: "Upper" or "Lower", the memory address accessed and the bounds, while with `handle nostop SIGSEGV` no additional information is displayed.

The usual output of a segfault is:

```
Program received signal SIGSEGV, Segmentation fault
0x0000000000400d7c in upper () at i386-mpx-sigsegv.c:68
68        value = *(p + len);
```

While a bound violation is presented as:

```
Program received signal SIGSEGV, Segmentation fault
Upper bound violation while accessing address 0x7fffffffc3b3
Bounds: [lower = 0x7fffffffc390, upper = 0x7fffffffc3a3]
0x0000000000400d7c in upper () at i386-mpx-sigsegv.c:68
68        value = *(p + len);
```

5.5 Stopping and Starting Multi-thread Programs

GDB supports debugging programs with multiple threads (see Section 4.10 [Debugging Programs with Multiple Threads], page 44). There are two modes of controlling execution of your program within the debugger. In the default mode, referred to as *all-stop mode*, when any thread in your program stops (for example, at a breakpoint or while being stepped), all other threads in the program are also stopped by GDB. On some targets, GDB also supports *non-stop mode*, in which other threads can continue to run freely while you examine the stopped thread in the debugger.

5.5.1 All-Stop Mode

In all-stop mode, whenever your program stops under GDB for any reason, *all* threads of execution stop, not just the current thread. This allows you to examine the overall state of the program, including switching between threads, without worrying that things may change underfoot.

Conversely, whenever you restart the program, *all* threads start executing. *This is true even when single-stepping* with commands like `step` or `next`.

In particular, GDB cannot single-step all threads in lockstep. Since thread scheduling is up to your debugging target's operating system (not controlled by GDB), other threads may execute more than one statement while the current thread completes a single step. Moreover, in general other threads stop in the middle of a statement, rather than at a clean statement boundary, when the program stops.

You might even find your program stopped in another thread after continuing or even single-stepping. This happens whenever some other thread runs into a breakpoint, a signal, or an exception before the first thread completes whatever you requested.

Whenever GDB stops your program, due to a breakpoint or a signal, it automatically selects the thread where that breakpoint or signal happened. GDB alerts you to the context switch with a message such as '`[Switching to Thread n]`' to identify the thread.

On some OSes, you can modify GDB's default behavior by locking the OS scheduler to allow only a single thread to run.

`set scheduler-locking` *mode*

Set the scheduler locking mode. It applies to normal execution, record mode, and replay mode. If it is `off`, then there is no locking and any thread may run at any time. If `on`, then only the current thread may run when the inferior is resumed. The `step` mode optimizes for single-stepping; it prevents other threads from preempting the current thread while you are stepping, so that the focus of debugging does not change unexpectedly. Other threads never get a chance to run when you step, and they are completely free to run when you use commands like '`continue`', '`until`', or '`finish`'. However, unless another thread hits a breakpoint during its timeslice, GDB does not change the current thread away from the thread that you are debugging. The `replay` mode behaves like `off` in record mode and like `on` in replay mode.

`show scheduler-locking`

Display the current scheduler locking mode.

By default, when you issue one of the execution commands such as `continue`, `next` or `step`, GDB allows only threads of the current inferior to run. For example, if GDB is attached to two inferiors, each with two threads, the `continue` command resumes only the two threads of the current inferior. This is useful, for example, when you debug a program that forks and you want to hold the parent stopped (so that, for instance, it doesn't run to exit), while you debug the child. In other situations, you may not be interested in inspecting the current state of any of the processes GDB is attached to, and you may want to resume them all until some breakpoint is hit. In the latter case, you can instruct GDB to allow all threads of all the inferiors to run with the `set schedule-multiple` command.

`set schedule-multiple`
: Set the mode for allowing threads of multiple processes to be resumed when an execution command is issued. When `on`, all threads of all processes are allowed to run. When `off`, only the threads of the current process are resumed. The default is `off`. The `scheduler-locking` mode takes precedence when set to `on`, or while you are stepping and set to `step`.

`show schedule-multiple`
: Display the current mode for resuming the execution of threads of multiple processes.

5.5.2 Non-Stop Mode

For some multi-threaded targets, GDB supports an optional mode of operation in which you can examine stopped program threads in the debugger while other threads continue to execute freely. This minimizes intrusion when debugging live systems, such as programs where some threads have real-time constraints or must continue to respond to external events. This is referred to as *non-stop* mode.

In non-stop mode, when a thread stops to report a debugging event, *only* that thread is stopped; GDB does not stop other threads as well, in contrast to the all-stop mode behavior. Additionally, execution commands such as `continue` and `step` apply by default only to the current thread in non-stop mode, rather than all threads as in all-stop mode. This allows you to control threads explicitly in ways that are not possible in all-stop mode — for example, stepping one thread while allowing others to run freely, stepping one thread while holding all others stopped, or stepping several threads independently and simultaneously.

To enter non-stop mode, use this sequence of commands before you run or attach to your program:

```
# If using the CLI, pagination breaks non-stop.
set pagination off

# Finally, turn it on!
set non-stop on
```

You can use these commands to manipulate the non-stop mode setting:

`set non-stop on`
: Enable selection of non-stop mode.

`set non-stop off`
: Disable selection of non-stop mode.

`show non-stop`
: Show the current non-stop enablement setting.

Note these commands only reflect whether non-stop mode is enabled, not whether the currently-executing program is being run in non-stop mode. In particular, the `set non-stop` preference is only consulted when GDB starts or connects to the target program, and it is generally not possible to switch modes once debugging has started. Furthermore, since not all targets support non-stop mode, even when you have enabled non-stop mode, GDB may still fall back to all-stop operation by default.

In non-stop mode, all execution commands apply only to the current thread by default. That is, `continue` only continues one thread. To continue all threads, issue `continue -a` or `c -a`.

You can use GDB's background execution commands (see Section 5.5.3 [Background Execution], page 92) to run some threads in the background while you continue to examine or step others from GDB. The MI execution commands (see Section 27.15 [GDB/MI Program Execution], page 574) are always executed asynchronously in non-stop mode.

Suspending execution is done with the `interrupt` command when running in the background, or *`Ctrl-c`* during foreground execution. In all-stop mode, this stops the whole process; but in non-stop mode the interrupt applies only to the current thread. To stop the whole program, use `interrupt -a`.

Other execution commands do not currently support the `-a` option.

In non-stop mode, when a thread stops, GDB doesn't automatically make that thread current, as it does in all-stop mode. This is because the thread stop notifications are asynchronous with respect to GDB's command interpreter, and it would be confusing if GDB unexpectedly changed to a different thread just as you entered a command to operate on the previously current thread.

5.5.3 Background Execution

GDB's execution commands have two variants: the normal foreground (synchronous) behavior, and a background (asynchronous) behavior. In foreground execution, GDB waits for the program to report that some thread has stopped before prompting for another command. In background execution, GDB immediately gives a command prompt so that you can issue other commands while your program runs.

If the target doesn't support async mode, GDB issues an error message if you attempt to use the background execution commands.

To specify background execution, add a `&` to the command. For example, the background form of the `continue` command is `continue&`, or just `c&`. The execution commands that accept background execution are:

`run` See Section 4.2 [Starting your Program], page 32.

`attach` See Section 4.7 [Debugging an Already-running Process], page 39.

`step` See Section 5.2 [Continuing and Stepping], page 80.

`stepi` See Section 5.2 [Continuing and Stepping], page 80.

`next` See Section 5.2 [Continuing and Stepping], page 80.

`nexti` See Section 5.2 [Continuing and Stepping], page 80.

`continue` See Section 5.2 [Continuing and Stepping], page 80.

`finish` See Section 5.2 [Continuing and Stepping], page 80.

`until` See Section 5.2 [Continuing and Stepping], page 80.

Background execution is especially useful in conjunction with non-stop mode for debugging programs with multiple threads; see Section 5.5.2 [Non-Stop Mode], page 91. However, you can also use these commands in the normal all-stop mode with the restriction that you

cannot issue another execution command until the previous one finishes. Examples of commands that are valid in all-stop mode while the program is running include `help` and `info break`.

You can interrupt your program while it is running in the background by using the `interrupt` command.

`interrupt`
`interrupt -a`

Suspend execution of the running program. In all-stop mode, `interrupt` stops the whole process, but in non-stop mode, it stops only the current thread. To stop the whole program in non-stop mode, use `interrupt -a`.

5.5.4 Thread-Specific Breakpoints

When your program has multiple threads (see Section 4.10 [Debugging Programs with Multiple Threads], page 44), you can choose whether to set breakpoints on all threads, or on a particular thread.

`break` *`location`* `thread` *`thread-id`*
`break` *`location`* `thread` *`thread-id`* `if ...`

location specifies source lines; there are several ways of writing them (see Section 9.2 [Specify Location], page 120), but the effect is always to specify some source line.

Use the qualifier '`thread` *`thread-id`*' with a breakpoint command to specify that you only want GDB to stop the program when a particular thread reaches this breakpoint. The *thread-id* specifier is one of the thread identifiers assigned by GDB, shown in the first column of the '`info threads`' display.

If you do not specify '`thread` *`thread-id`*' when you set a breakpoint, the breakpoint applies to *all* threads of your program.

You can use the `thread` qualifier on conditional breakpoints as well; in this case, place '`thread` *`thread-id`*' before or after the breakpoint condition, like this:

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

Thread-specific breakpoints are automatically deleted when GDB detects the corresponding thread is no longer in the thread list. For example:

```
(gdb) c
Thread-specific breakpoint 3 deleted - thread 28 no longer in the thread list.
```

There are several ways for a thread to disappear, such as a regular thread exit, but also when you detach from the process with the `detach` command (see Section 4.7 [Debugging an Already-running Process], page 39), or if GDB loses the remote connection (see Chapter 20 [Remote Debugging], page 299), etc. Note that with some targets, GDB is only able to detect a thread has exited when the user explictly asks for the thread list with the `info threads` command.

5.5.5 Interrupted System Calls

There is an unfortunate side effect when using GDB to debug multi-threaded programs. If one thread stops for a breakpoint, or for some other reason, and another thread is blocked

in a system call, then the system call may return prematurely. This is a consequence of the interaction between multiple threads and the signals that GDB uses to implement breakpoints and other events that stop execution.

To handle this problem, your program should check the return value of each system call and react appropriately. This is good programming style anyways.

For example, do not write code like this:

```
sleep (10);
```

The call to `sleep` will return early if a different thread stops at a breakpoint or for some other reason.

Instead, write this:

```
int unslept = 10;
while (unslept > 0)
  unslept = sleep (unslept);
```

A system call is allowed to return early, so the system is still conforming to its specification. But GDB does cause your multi-threaded program to behave differently than it would without GDB.

Also, GDB uses internal breakpoints in the thread library to monitor certain events such as thread creation and thread destruction. When such an event happens, a system call in another thread may return prematurely, even though your program does not appear to stop.

5.5.6 Observer Mode

If you want to build on non-stop mode and observe program behavior without any chance of disruption by GDB, you can set variables to disable all of the debugger's attempts to modify state, whether by writing memory, inserting breakpoints, etc. These operate at a low level, intercepting operations from all commands.

When all of these are set to `off`, then GDB is said to be *observer mode*. As a convenience, the variable `observer` can be set to disable these, plus enable non-stop mode.

Note that GDB will not prevent you from making nonsensical combinations of these settings. For instance, if you have enabled `may-insert-breakpoints` but disabled `may-write-memory`, then breakpoints that work by writing trap instructions into the code stream will still not be able to be placed.

`set observer on`
`set observer off`
: When set to `on`, this disables all the permission variables below (except for `insert-fast-tracepoints`), plus enables non-stop debugging. Setting this to `off` switches back to normal debugging, though remaining in non-stop mode.

`show observer`
: Show whether observer mode is on or off.

`set may-write-registers on`
`set may-write-registers off`
: This controls whether GDB will attempt to alter the values of registers, such as with assignment expressions in `print`, or the `jump` command. It defaults to `on`.

`show may-write-registers`
: Show the current permission to write registers.

`set may-write-memory on`
`set may-write-memory off`
: This controls whether GDB will attempt to alter the contents of memory, such as with assignment expressions in `print`. It defaults to `on`.

`show may-write-memory`
: Show the current permission to write memory.

`set may-insert-breakpoints on`
`set may-insert-breakpoints off`
: This controls whether GDB will attempt to insert breakpoints. This affects all breakpoints, including internal breakpoints defined by GDB. It defaults to `on`.

`show may-insert-breakpoints`
: Show the current permission to insert breakpoints.

`set may-insert-tracepoints on`
`set may-insert-tracepoints off`
: This controls whether GDB will attempt to insert (regular) tracepoints at the beginning of a tracing experiment. It affects only non-fast tracepoints, fast tracepoints being under the control of `may-insert-fast-tracepoints`. It defaults to `on`.

`show may-insert-tracepoints`
: Show the current permission to insert tracepoints.

`set may-insert-fast-tracepoints on`
`set may-insert-fast-tracepoints off`
: This controls whether GDB will attempt to insert fast tracepoints at the beginning of a tracing experiment. It affects only fast tracepoints, regular (non-fast) tracepoints being under the control of `may-insert-tracepoints`. It defaults to `on`.

`show may-insert-fast-tracepoints`
: Show the current permission to insert fast tracepoints.

`set may-interrupt on`
`set may-interrupt off`
: This controls whether GDB will attempt to interrupt or stop program execution. When this variable is `off`, the `interrupt` command will have no effect, nor will *`Ctrl-c`*. It defaults to `on`.

`show may-interrupt`
: Show the current permission to interrupt or stop the program.

6 Running programs backward

When you are debugging a program, it is not unusual to realize that you have gone too far, and some event of interest has already happened. If the target environment supports it, GDB can allow you to "rewind" the program by running it backward.

A target environment that supports reverse execution should be able to "undo" the changes in machine state that have taken place as the program was executing normally. Variables, registers etc. should revert to their previous values. Obviously this requires a great deal of sophistication on the part of the target environment; not all target environments can support reverse execution.

When a program is executed in reverse, the instructions that have most recently been executed are "un-executed", in reverse order. The program counter runs backward, following the previous thread of execution in reverse. As each instruction is "un-executed", the values of memory and/or registers that were changed by that instruction are reverted to their previous states. After executing a piece of source code in reverse, all side effects of that code should be "undone", and all variables should be returned to their prior values[1].

On some platforms, GDB has built-in support for reverse execution, activated with the `record` or `record btrace` commands. See Chapter 7 [Process Record and Replay], page 99. Some remote targets, typically full system emulators, support reverse execution directly without requiring any special command.

If you are debugging in a target environment that supports reverse execution, GDB provides the following commands.

`reverse-continue` [*ignore-count*]
`rc` [*ignore-count*]
: Beginning at the point where your program last stopped, start executing in reverse. Reverse execution will stop for breakpoints and synchronous exceptions (signals), just like normal execution. Behavior of asynchronous signals depends on the target environment.

`reverse-step` [*count*]
: Run the program backward until control reaches the start of a different source line; then stop it, and return control to GDB.

  Like the `step` command, `reverse-step` will only stop at the beginning of a source line. It "un-executes" the previously executed source line. If the previous source line included calls to debuggable functions, `reverse-step` will step (backward) into the called function, stopping at the beginning of the *last* statement in the called function (typically a return statement).

  Also, as with the `step` command, if non-debuggable functions are called, `reverse-step` will run thru them backward without stopping.

[1] Note that some side effects are easier to undo than others. For instance, memory and registers are relatively easy, but device I/O is hard. Some targets may be able undo things like device I/O, and some may not.

The contract between GDB and the reverse executing target requires only that the target do something reasonable when GDB tells it to execute backwards, and then report the results back to GDB. Whatever the target reports back to GDB, GDB will report back to the user. GDB assumes that the memory and registers that the target reports are in a consistent state, but GDB accepts whatever it is given.

`reverse-stepi` [*count*]
: Reverse-execute one machine instruction. Note that the instruction to be reverse-executed is *not* the one pointed to by the program counter, but the instruction executed prior to that one. For instance, if the last instruction was a jump, `reverse-stepi` will take you back from the destination of the jump to the jump instruction itself.

`reverse-next` [*count*]
: Run backward to the beginning of the previous line executed in the current (innermost) stack frame. If the line contains function calls, they will be "un-executed" without stopping. Starting from the first line of a function, `reverse-next` will take you back to the caller of that function, *before* the function was called, just as the normal `next` command would take you from the last line of a function back to its return to its caller[2].

`reverse-nexti` [*count*]
: Like `nexti`, `reverse-nexti` executes a single instruction in reverse, except that called functions are "un-executed" atomically. That is, if the previously executed instruction was a return from another function, `reverse-nexti` will continue to execute in reverse until the call to that function (from the current stack frame) is reached.

`reverse-finish`
: Just as the `finish` command takes you to the point where the current function returns, `reverse-finish` takes you to the point where it was called. Instead of ending up at the end of the current function invocation, you end up at the beginning.

`set exec-direction`
: Set the direction of target execution.

`set exec-direction reverse`
: GDB will perform all execution commands in reverse, until the exec-direction mode is changed to "forward". Affected commands include `step, stepi, next, nexti, continue, and finish`. The `return` command cannot be used in reverse mode.

`set exec-direction forward`
: GDB will perform all execution commands in the normal fashion. This is the default.

[2] Unless the code is too heavily optimized.

7 Recording Inferior's Execution and Replaying It

On some platforms, GDB provides a special *process record and replay* target that can record a log of the process execution, and replay it later with both forward and reverse execution commands.

When this target is in use, if the execution log includes the record for the next instruction, GDB will debug in *replay mode*. In the replay mode, the inferior does not really execute code instructions. Instead, all the events that normally happen during code execution are taken from the execution log. While code is not really executed in replay mode, the values of registers (including the program counter register) and the memory of the inferior are still changed as they normally would. Their contents are taken from the execution log.

If the record for the next instruction is not in the execution log, GDB will debug in *record mode*. In this mode, the inferior executes normally, and GDB records the execution log for future replay.

The process record and replay target supports reverse execution (see Chapter 6 [Reverse Execution], page 97), even if the platform on which the inferior runs does not. However, the reverse execution is limited in this case by the range of the instructions recorded in the execution log. In other words, reverse execution on platforms that don't support it directly can only be done in the replay mode.

When debugging in the reverse direction, GDB will work in replay mode as long as the execution log includes the record for the previous instruction; otherwise, it will work in record mode, if the platform supports reverse execution, or stop if not.

Currently, process record and replay is supported on ARM, Aarch64, Moxie, PowerPC, PowerPC64, S/390, and x86 (i386/amd64) running GNU/Linux. Process record and replay can be used both when native debugging, and when remote debugging via `gdbserver`.

For architecture environments that support process record and replay, GDB provides the following commands:

`record` *`method`*

This command starts the process record and replay target. The recording method can be specified as parameter. Without a parameter the command uses the `full` recording method. The following recording methods are available:

`full`
: Full record/replay recording using GDB's software record and replay implementation. This method allows replaying and reverse execution.

`btrace` *`format`*
: Hardware-supported instruction recording, supported on Intel processors. This method does not record data. Further, the data is collected in a ring buffer so old data will be overwritten when the buffer is full. It allows limited reverse execution. Variables and registers are not available during reverse execution. In remote debugging, recording continues on disconnect. Recorded data can be inspected after reconnecting. The recording may be stopped using `record stop`.

The recording format can be specified as parameter. Without a parameter the command chooses the recording format. The following recording formats are available:

`bts`
: Use the *Branch Trace Store* (BTS) recording format. In this format, the processor stores a from/to record for each executed branch in the btrace ring buffer.

`pt`
: Use the *Intel Processor Trace* recording format. In this format, the processor stores the execution trace in a compressed form that is afterwards decoded by GDB.

  The trace can be recorded with very low overhead. The compressed trace format also allows small trace buffers to already contain a big number of instructions compared to BTS.

  Decoding the recorded execution trace, on the other hand, is more expensive than decoding BTS trace. This is mostly due to the increased number of instructions to process. You should increase the buffer-size with care.

Not all recording formats may be available on all processors.

The process record and replay target can only debug a process that is already running. Therefore, you need first to start the process with the *`run`* or *`start`* commands, and then start the recording with the *`record method`* command.

Displaced stepping (see Appendix D [displaced stepping], page 671) will be automatically disabled when process record and replay target is started. That's because the process record and replay target doesn't support displaced stepping.

If the inferior is in the non-stop mode (see Section 5.5.2 [Non-Stop Mode], page 91) or in the asynchronous execution mode (see Section 5.5.3 [Background Execution], page 92), not all recording methods are available. The `full` recording method does not support these two modes.

`record stop`

Stop the process record and replay target. When process record and replay target stops, the entire execution log will be deleted and the inferior will either be terminated, or will remain in its final state.

When you stop the process record and replay target in record mode (at the end of the execution log), the inferior will be stopped at the next instruction that would have been recorded. In other words, if you record for a while and then stop recording, the inferior process will be left in the same state as if the recording never happened.

On the other hand, if the process record and replay target is stopped while in replay mode (that is, not at the end of the execution log, but at some earlier point), the inferior process will become "live" at that earlier state, and it will then be possible to continue the usual "live" debugging of the process from that state.

When the inferior process exits, or GDB detaches from it, process record and replay target will automatically stop itself.

`record goto`
: Go to a specific location in the execution log. There are several ways to specify the location to go to:

  `record goto begin`
  `record goto start`
  : Go to the beginning of the execution log.

  `record goto end`
  : Go to the end of the execution log.

  `record goto` *n*
  : Go to instruction number *n* in the execution log.

`record save` *filename*
: Save the execution log to a file *filename*. Default filename is `gdb_record.`*process_id*, where *process_id* is the process ID of the inferior.

  This command may not be available for all recording methods.

`record restore` *filename*
: Restore the execution log from a file *filename*. File must have been created with `record save`.

`set record full insn-number-max` *limit*
`set record full insn-number-max unlimited`
: Set the limit of instructions to be recorded for the `full` recording method. Default value is 200000.

  If *limit* is a positive number, then GDB will start deleting instructions from the log once the number of the record instructions becomes greater than *limit*. For every new recorded instruction, GDB will delete the earliest recorded instruction to keep the number of recorded instructions at the limit. (Since deleting recorded instructions loses information, GDB lets you control what happens when the limit is reached, by means of the `stop-at-limit` option, described below.)

  If *limit* is `unlimited` or zero, GDB will never delete recorded instructions from the execution log. The number of recorded instructions is limited only by the available memory.

`show record full insn-number-max`
: Show the limit of instructions to be recorded with the `full` recording method.

`set record full stop-at-limit`
: Control the behavior of the `full` recording method when the number of recorded instructions reaches the limit. If ON (the default), GDB will stop when the limit is reached for the first time and ask you whether you want to stop the inferior or continue running it and recording the execution log. If you decide to continue recording, each new recorded instruction will cause the oldest one to be deleted.

  If this option is OFF, GDB will automatically delete the oldest record to make room for each new one, without asking.

`show record full stop-at-limit`
: Show the current setting of `stop-at-limit`.

`set record full memory-query`

Control the behavior when GDB is unable to record memory changes caused by an instruction for the `full` recording method. If ON, GDB will query whether to stop the inferior in that case.

If this option is OFF (the default), GDB will automatically ignore the effect of such instructions on memory. Later, when GDB replays this execution log, it will mark the log of this instruction as not accessible, and it will not affect the replay results.

`show record full memory-query`

Show the current setting of `memory-query`.

The `btrace` record target does not trace data. As a convenience, when replaying, GDB reads read-only memory off the live program directly, assuming that the addresses of the read-only areas don't change. This for example makes it possible to disassemble code while replaying, but not to print variables. In some cases, being able to inspect variables might be useful. You can use the following command for that:

`set record btrace replay-memory-access`

Control the behavior of the `btrace` recording method when accessing memory during replay. If `read-only` (the default), GDB will only allow accesses to read-only memory. If `read-write`, GDB will allow accesses to read-only and to read-write memory. Beware that the accessed memory corresponds to the live target and not necessarily to the current replay position.

`set record btrace cpu` *`identifier`*

Set the processor to be used for enabling workarounds for processor errata when decoding the trace.

Processor errata are defects in processor operation, caused by its design or manufacture. They can cause a trace not to match the specification. This, in turn, may cause trace decode to fail. GDB can detect erroneous trace packets and correct them, thus avoiding the decoding failures. These corrections are known as *errata workarounds*, and are enabled based on the processor on which the trace was recorded.

By default, GDB attempts to detect the processor automatically, and apply the necessary workarounds for it. However, you may need to specify the processor if GDB does not yet support it. This command allows you to do that, and also allows to disable the workarounds.

The argument *identifier* identifies the CPU and is of the form: *`vendor:processor identifier`*. In addition, there are two special identifiers, `none` and `auto` (default).

The following vendor identifiers and corresponding processor identifiers are currently supported:

`intel` *family*/*model*[/*stepping*]

On GNU/Linux systems, the processor *family*, *model*, and *stepping* can be obtained from `/proc/cpuinfo`.

If *identifier* is `auto`, enable errata workarounds for the processor on which the trace was recorded. If *identifier* is `none`, errata workarounds are disabled.

For example, when using an old GDB on a new system, decode may fail because GDB does not support the new processor. It often suffices to specify an older processor that GDB supports.

```
(gdb) info record
Active record target: record-btrace
Recording format: Intel Processor Trace.
Buffer size: 16kB.
Failed to configure the Intel Processor Trace decoder: unknown cpu.
(gdb) set record btrace cpu intel:6/158
(gdb) info record
Active record target: record-btrace
Recording format: Intel Processor Trace.
Buffer size: 16kB.
Recorded 84872 instructions in 3189 functions (0 gaps) for thread 1 (...).
```

`show record btrace replay-memory-access`
: Show the current setting of `replay-memory-access`.

`show record btrace cpu`
: Show the processor to be used for enabling trace decode errata workarounds.

`set record btrace bts buffer-size` *size*
`set record btrace bts buffer-size unlimited`
: Set the requested ring buffer size for branch tracing in BTS format. Default is 64KB.

  If *size* is a positive number, then GDB will try to allocate a buffer of at least *size* bytes for each new thread that uses the btrace recording method and the BTS format. The actually obtained buffer size may differ from the requested *size*. Use the `info record` command to see the actual buffer size for each thread that uses the btrace recording method and the BTS format.

  If *limit* is `unlimited` or zero, GDB will try to allocate a buffer of 4MB.

  Bigger buffers mean longer traces. On the other hand, GDB will also need longer to process the branch trace data before it can be used.

`show record btrace bts buffer-size` *size*
: Show the current setting of the requested ring buffer size for branch tracing in BTS format.

`set record btrace pt buffer-size` *size*
`set record btrace pt buffer-size unlimited`
: Set the requested ring buffer size for branch tracing in Intel Processor Trace format. Default is 16KB.

  If *size* is a positive number, then GDB will try to allocate a buffer of at least *size* bytes for each new thread that uses the btrace recording method and the Intel Processor Trace format. The actually obtained buffer size may differ from the requested *size*. Use the `info record` command to see the actual buffer size for each thread.

If *limit* is `unlimited` or zero, GDB will try to allocate a buffer of 4MB.

Bigger buffers mean longer traces. On the other hand, GDB will also need longer to process the branch trace data before it can be used.

`show record btrace pt buffer-size` *size*
: Show the current setting of the requested ring buffer size for branch tracing in Intel Processor Trace format.

`info record`
: Show various statistics about the recording depending on the recording method:

  `full` For the `full` recording method, it shows the state of process record and its in-memory execution log buffer, including:

  - Whether in record mode or replay mode.
  - Lowest recorded instruction number (counting from when the current execution log started recording instructions).
  - Highest recorded instruction number.
  - Current instruction about to be replayed (if in replay mode).
  - Number of instructions contained in the execution log.
  - Maximum number of instructions that may be contained in the execution log.

  `btrace` For the `btrace` recording method, it shows:

  - Recording format.
  - Number of instructions that have been recorded.
  - Number of blocks of sequential control-flow formed by the recorded instructions.
  - Whether in record mode or replay mode.

  For the `bts` recording format, it also shows:

  - Size of the perf ring buffer.

  For the `pt` recording format, it also shows:

  - Size of the perf ring buffer.

`record delete`
: When record target runs in replay mode ("in the past"), delete the subsequent execution log and begin to record a new execution log starting from the current address. This means you will abandon the previously recorded "future" and begin recording a new "future".

`record instruction-history`
: Disassembles instructions from the recorded execution log. By default, ten instructions are disassembled. This can be changed using the `set record instruction-history-size` command. Instructions are printed in execution order.

  It can also print mixed source+disassembly if you specify the the `/m` or `/s` modifier, and print the raw instructions in hex as well as in symbolic form by specifying the `/r` modifier.

The current position marker is printed for the instruction at the current program counter value. This instruction can appear multiple times in the trace and the current position marker will be printed every time. To omit the current position marker, specify the `/p` modifier.

To better align the printed instructions when the trace contains instructions from more than one function, the function name may be omitted by specifying the `/f` modifier.

Speculatively executed instructions are prefixed with '`?`'. This feature is not available for all recording formats.

There are several ways to specify what part of the execution log to disassemble:

`record instruction-history` *`insn`*
: Disassembles ten instructions starting from instruction number *insn*.

`record instruction-history` *`insn`*`, +/-`*`n`*
: Disassembles *n* instructions around instruction number *insn*. If *n* is preceded with `+`, disassembles *n* instructions after instruction number *insn*. If *n* is preceded with `-`, disassembles *n* instructions before instruction number *insn*.

`record instruction-history`
: Disassembles ten more instructions after the last disassembly.

`record instruction-history -`
: Disassembles ten more instructions before the last disassembly.

`record instruction-history` *`begin`*`, `*`end`*
: Disassembles instructions beginning with instruction number *begin* until instruction number *end*. The instruction number *end* is included.

This command may not be available for all recording methods.

`set record instruction-history-size` *`size`*
`set record instruction-history-size unlimited`
: Define how many instructions to disassemble in the `record instruction-history` command. The default value is 10. A *size* of `unlimited` means unlimited instructions.

`show record instruction-history-size`
: Show how many instructions to disassemble in the `record instruction-history` command.

`record function-call-history`
: Prints the execution history at function granularity. For each sequence of instructions that belong to the same function, it prints the name of that function, the source lines for this instruction sequence (if the `/l` modifier is specified), and the instructions numbers that form the sequence (if the `/i` modifier is specified). The function names are indented to reflect the call stack depth if the `/c` modifier is specified. The `/l`, `/i`, and `/c` modifiers can be given together.

```
(gdb) list 1, 10
```

```
1   void foo (void)
2   {
3   }
4
5   void bar (void)
6   {
7     ...
8     foo ();
9     ...
10  }
(gdb) record function-call-history /ilc
1  bar     inst 1,4     at foo.c:6,8
2    foo   inst 5,10    at foo.c:2,3
3  bar     inst 11,13   at foo.c:9,10
```

By default, ten functions are printed. This can be changed using the `set record function-call-history-size` command. Functions are printed in execution order. There are several ways to specify what to print:

`record function-call-history` *`func`*
: Prints ten functions starting from function number *func*.

`record function-call-history` *`func`*`, +/-`*`n`*
: Prints *n* functions around function number *func*. If *n* is preceded with `+`, prints *n* functions after function number *func*. If *n* is preceded with `-`, prints *n* functions before function number *func*.

`record function-call-history`
: Prints ten more functions after the last ten-function print.

`record function-call-history -`
: Prints ten more functions before the last ten-function print.

`record function-call-history` *`begin`*`,` *`end`*
: Prints functions beginning with function number *begin* until function number *end*. The function number *end* is included.

This command may not be available for all recording methods.

`set record function-call-history-size` *`size`*
`set record function-call-history-size unlimited`
: Define how many functions to print in the `record function-call-history` command. The default value is 10. A size of `unlimited` means unlimited functions.

`show record function-call-history-size`
: Show how many functions to print in the `record function-call-history` command.

8 Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*.

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is *selected* by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in. See Section 8.3 [Selecting a Frame], page 111.

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command (see Section 8.4 [Information about a Frame], page 113).

8.1 Stack Frames

The call stack is divided up into contiguous pieces called *stack frames*, or *frames* for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function `main`. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one byte whose address serves as the address of the frame. Usually this address is kept in a register called the *frame pointer register* (see Section 10.14 [Registers], page 169) while execution is going on in that frame.

GDB labels each existing stack frame with a *level*, a number that is zero for the innermost frame, one for the frame that called it, and so on upward. These level numbers give you a way of designating stack frames in GDB commands. The terms *frame number* and *frame level* can be used interchangeably to describe this number.

Some compilers provide a way to compile functions so that they operate without stack frames. (For example, the GCC option

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

generates functions without a frame.) This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with

these function invocations. If the innermost function invocation has no stack frame, GDB nevertheless regards it as though it had a separate frame, which is numbered zero as usual, allowing correct tracing of the function call chain. However, GDB has no provision for frameless functions elsewhere in the stack.

8.2 Backtraces

A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

To print a backtrace of the entire stack, use the `backtrace` command, or its alias `bt`. This command will print one line per frame for frames in the stack. By default, all stack frames are printed. You can stop the backtrace at any time by typing the system interrupt character, normally *Ctrl-c*.

`backtrace [`*option*`]... [`*qualifier*`]... [`*count*`]`
`bt [`*option*`]... [`*qualifier*`]... [`*count*`]`

Print the backtrace of the entire stack.

The optional *count* can be one of the following:

n
n Print only the innermost *n* frames, where *n* is a positive number.

-n
-n Print only the outermost *n* frames, where *n* is a positive number.

Options:

`-full` Print the values of the local variables also. This can be combined with the optional *count* to limit the number of frames shown.

`-no-filters`
Do not run Python frame filters on this backtrace. See Section 23.3.2.9 [Frame Filter API], page 402, for more information. Additionally use [disable frame-filter all], page 116, to turn off all frame filters. This is only relevant when GDB has been configured with `Python` support.

`-hide` A Python frame filter might decide to "elide" some frames. Normally such elided frames are still printed, but they are indented relative to the filtered frames that cause them to be elided. The `-hide` option causes elided frames to not be printed at all.

The `backtrace` command also supports a number of options that allow overriding relevant global print settings as set by `set backtrace` and `set print` subcommands:

`-past-main [on|off]`
Set whether backtraces should continue past `main`. Related setting: [set backtrace past-main], page 110.

`-past-entry [on|off]`
Set whether backtraces should continue past the entry point of a program. Related setting: [set backtrace past-entry], page 110.

`-entry-values no|only|preferred|if-needed|both|compact|default`
: Set printing of function arguments at function entry. Related setting: [set print entry-values], page 152.

`-frame-arguments all|scalars|none`
: Set printing of non-scalar frame arguments. Related setting: [set print frame-arguments], page 151.

`-raw-frame-arguments [on|off]`
: Set whether to print frame arguments in raw form. Related setting: [set print raw-frame-arguments], page 152.

`-frame-info auto|source-line|location|source-and-location|location-and-address|short-location`
: Set printing of frame information. Related setting: [set print frame-info], page 154.

The optional *qualifier* is maintained for backward compatibility. It can be one of the following:

`full` Equivalent to the `-full` option.

`no-filters`
: Equivalent to the `-no-filters` option.

`hide` Equivalent to the `-hide` option.

The names `where` and `info stack` (abbreviated `info s`) are additional aliases for `backtrace`.

In a multi-threaded program, GDB by default shows the backtrace only for the current thread. To display the backtrace for several or all of the threads, use the command `thread apply` (see Section 4.10 [Threads], page 44). For example, if you type *`thread apply all backtrace`*, GDB will display the backtrace for all the threads; this is handy when you debug a core dump of a multi-threaded program.

Each line in the backtrace shows the frame number and the function name. The program counter value is also shown—unless you use `set print address off`. The backtrace also shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number.

Here is an example of a backtrace. It was made with the command '`bt 3`', so it shows the innermost three frames.

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

The display for frame zero does not begin with a program counter value, indicating that your program has stopped at the beginning of the code for line `993` of `builtin.c`.

The value of parameter `data` in frame 1 has been replaced by `...`. By default, GDB prints the value of a parameter only if it is a scalar (integer, pointer, enumeration, etc). See command *`set print frame-arguments`* in Section 10.9 [Print Settings], page 149, for more

details on how to configure the way function parameter values are printed. The command *set print frame-info* (see Section 10.9 [Print Settings], page 149) controls what frame information is printed.

If your program was compiled with optimizations, some compilers will optimize away arguments passed to functions if those arguments are never used after the call. Such optimizations generate code that passes arguments through registers, but doesn't store those arguments in the stack frame. GDB has no way of displaying such arguments in stack frames other than the innermost one. Here's what such a backtrace might look like:

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

The values of arguments that were not saved in their stack frames are shown as '`<optimized out>`'.

If you need to display the values of such optimized-out arguments, either deduce that from other variables whose values depend on the one you are interested in, or recompile without optimizations.

Most programs have a standard user entry point—a place where system libraries and startup code transition into user code. For C this is `main`[1]. When GDB finds the entry function in a backtrace it will terminate the backtrace, to avoid tracing into highly system-specific (and generally uninteresting) code.

If you need to examine the startup code, or limit the number of levels in a backtrace, you can change this behavior:

`set backtrace past-main`
`set backtrace past-main on`
: Backtraces will continue past the user entry point.

`set backtrace past-main off`
: Backtraces will stop when they encounter the user entry point. This is the default.

`show backtrace past-main`
: Display the current user entry point backtrace policy.

`set backtrace past-entry`
`set backtrace past-entry on`
: Backtraces will continue past the internal entry point of an application. This entry point is encoded by the linker when the application is built, and is likely before the user entry point `main` (or equivalent) is called.

`set backtrace past-entry off`
: Backtraces will stop when they encounter the internal entry point of an application. This is the default.

[1] Note that embedded programs (the so-called "free-standing" environment) are not required to have a `main` function as the entry point. They could even have multiple entry points.

`show backtrace past-entry`
: Display the current internal entry point backtrace policy.

`set backtrace limit` *n*
`set backtrace limit 0`
`set backtrace limit unlimited`
: Limit the backtrace to *n* levels. A value of `unlimited` or zero means unlimited levels.

`show backtrace limit`
: Display the current limit on backtrace levels.

You can control how file names are displayed.

`set filename-display`
`set filename-display relative`
: Display file names relative to the compilation directory. This is the default.

`set filename-display basename`
: Display only basename of a filename.

`set filename-display absolute`
: Display an absolute filename.

`show filename-display`
: Show the current way to display filenames.

8.3 Selecting a Frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

`frame` [*frame-selection-spec*]
`f` [*frame-selection-spec*]
: The `frame` command allows different stack frames to be selected. The *frame-selection-spec* can be any of the following:

  num

  `level` *num*
  : Select frame level *num*. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest level frame is usually the one for `main`.

    As this is the most common method of navigating the frame stack, the string `level` can be omitted. For example, the following two commands are equivalent:

    ```
    (gdb) frame 3
    (gdb) frame level 3
    ```

  `address` *stack-address*
  : Select the frame with stack address *stack-address*. The *stack-address* for a frame can be seen in the output of `info frame`, for example:

    ```
    (gdb) info frame
    ```

```
Stack level 1, frame at 0x7fffffffda30:
 rip = 0x40066d in b (amd64-entry-value.cc:59); saved rip 0x4004c5
 tail call frame, caller of frame at 0x7fffffffda30
 source language c++.
 Arglist at unknown address.
 Locals at unknown address, Previous frame's sp is 0x7fffffffda30
```

The *stack-address* for this frame is `0x7fffffffda30` as indicated by the line:

```
Stack level 1, frame at 0x7fffffffda30:
```

`function` *function-name*
: Select the stack frame for function *function-name*. If there are multiple stack frames for function *function-name* then the inner most stack frame is selected.

`view` *stack-address* [*pc-addr*]
: View a frame that is not part of GDB's backtrace. The frame viewed has stack address *stack-addr*, and optionally, a program counter address of *pc-addr*.

  This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful when your program has multiple stacks and switches between them.

  When viewing a frame outside the current backtrace using `frame view` then you can always return to the original stack using one of the previous stack frame selection instructions, for example `frame level 0`.

`up` *n*
: Move *n* frames up the stack; *n* defaults to 1. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer.

`down` *n*
: Move *n* frames down the stack; *n* defaults to 1. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. You may abbreviate `down` as `do`.

All of these commands end by printing two lines of output describing the frame. The first line shows the frame number, the function name, the arguments, and the source file and line number of execution in that frame. The second line shows the text of that source line.

For example:

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

After such a printout, the `list` command with no arguments prints ten lines centered on the point of execution in the frame. You can also edit the program at the point of execution with your favorite editing program by typing `edit`. See Section 9.1 [Printing Source Lines], page 119, for details.

`select-frame` [*frame-selection-spec*]
: The `select-frame` command is a variant of `frame` that does not display the new frame after selecting it. This command is intended primarily for use in GDB command scripts, where the output might be unnecessary and distracting. The *frame-selection-spec* is as for the `frame` command described in Section 8.3 [Selecting a Frame], page 111.

`up-silently` *n*
`down-silently` *n*
: These two commands are variants of `up` and `down`, respectively; they differ in that they do their work silently, without causing display of the new frame. They are intended primarily for use in GDB command scripts, where the output might be unnecessary and distracting.

8.4 Information About a Frame

There are several other commands to print information about the selected stack frame.

`frame`
`f`
: When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated `f`. With an argument, this command is used to select a stack frame. See Section 8.3 [Selecting a Frame], page 111.

`info frame`
`info f`
: This command prints a verbose description of the selected stack frame, including:
  - the address of the frame
  - the address of the next frame down (called by this frame)
  - the address of the next frame up (caller of this frame)
  - the language in which the source code corresponding to this frame is written
  - the address of the frame's arguments
  - the address of the frame's local variables
  - the program counter saved in it (the address of execution in the caller frame)
  - which registers were saved in the frame

  The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

`info frame` [*frame-selection-spec*]
`info f` [*frame-selection-spec*]
: Print a verbose description of the frame selected by *frame-selection-spec*. The *frame-selection-spec* is the same as for the `frame` command (see Section 8.3 [Selecting a Frame], page 111). The selected frame remains unchanged by this command.

`info args [-q]`
: Print the arguments of the selected frame, each on a separate line.

The optional flag '`-q`', which stands for '`quiet`', disables printing header information and messages explaining why no argument have been printed.

`info args [-q] [-t type_regexp] [regexp]`

Like *`info args`*, but only print the arguments selected with the provided regexp(s).

If *regexp* is provided, print only the arguments whose names match the regular expression *regexp*.

If *type_regexp* is provided, print only the arguments whose types, as printed by the `whatis` command, match the regular expression *type_regexp*. If *type_regexp* contains space(s), it should be enclosed in quote characters. If needed, use backslash to escape the meaning of special characters or quotes.

If both *regexp* and *type_regexp* are provided, an argument is printed only if its name matches *regexp* and its type matches *type_regexp*.

`info locals [-q]`

Print the local variables of the selected frame, each on a separate line. These are all variables (declared either static or automatic) accessible at the point of execution of the selected frame.

The optional flag '`-q`', which stands for '`quiet`', disables printing header information and messages explaining why no local variables have been printed.

`info locals [-q] [-t type_regexp] [regexp]`

Like *`info locals`*, but only print the local variables selected with the provided regexp(s).

If *regexp* is provided, print only the local variables whose names match the regular expression *regexp*.

If *type_regexp* is provided, print only the local variables whose types, as printed by the `whatis` command, match the regular expression *type_regexp*. If *type_regexp* contains space(s), it should be enclosed in quote characters. If needed, use backslash to escape the meaning of special characters or quotes.

If both *regexp* and *type_regexp* are provided, a local variable is printed only if its name matches *regexp* and its type matches *type_regexp*.

The command *`info locals -q -t type_regexp`* can usefully be combined with the commands *`frame apply`* and *`thread apply`*. For example, your program might use Resource Acquisition Is Initialization types (RAII) such as `lock_something_t`: each local variable of type `lock_something_t` automatically places a lock that is destroyed when the variable goes out of scope. You can then list all acquired locks in your program by doing

```
thread apply all -s frame apply all -s info locals -q -t lock_something_t
```

or the equivalent shorter form

```
tfaas i lo -q -t lock_something_t
```

8.5 Applying a Command to Several Frames.

`frame apply [all | count | -count | level level...] [option]... command`

The `frame apply` command allows you to apply the named *command* to one or more frames.

`all` Specify `all` to apply *command* to all frames.

`count` Use *count* to apply *command* to the innermost *count* frames, where *count* is a positive number.

`-count` Use *-count* to apply *command* to the outermost *count* frames, where *count* is a positive number.

`level` Use `level` to apply *command* to the set of frames identified by the *level* list. *level* is a frame level or a range of frame levels as *level1-level2*. The frame level is the number shown in the first field of the '`backtrace`' command output. E.g., '`2-4 6-8 3`' indicates to apply *command* for the frames at levels 2, 3, 4, 6, 7, 8, and then again on frame at level 3.

Note that the frames on which `frame apply` applies a command are also influenced by the `set backtrace` settings such as `set backtrace past-main` and `set backtrace limit N`. See Section 8.2 [Backtraces], page 108.

The `frame apply` command also supports a number of options that allow overriding relevant `set backtrace` settings:

`-past-main [on|off]`
Whether backtraces should continue past `main`. Related setting: [set backtrace past-main], page 110.

`-past-entry [on|off]`
Whether backtraces should continue past the entry point of a program. Related setting: [set backtrace past-entry], page 110.

By default, GDB displays some frame information before the output produced by *command*, and an error raised during the execution of a *command* will abort `frame apply`. The following options can be used to fine-tune these behaviors:

`-c` The flag `-c`, which stands for '`continue`', causes any errors in *command* to be displayed, and the execution of `frame apply` then continues.

`-s` The flag `-s`, which stands for '`silent`', causes any errors or empty output produced by a *command* to be silently ignored. That is, the execution continues, but the frame information and errors are not printed.

`-q` The flag `-q` ('`quiet`') disables printing the frame information.

The following example shows how the flags `-c` and `-s` are working when applying the command `p j` to all frames, where variable `j` can only be successfully printed in the outermost `#1 main` frame.

```
(gdb) frame apply all p j
#0  some_function (i=5) at fun.c:4
No symbol "j" in current context.
(gdb) frame apply all -c p j
#0  some_function (i=5) at fun.c:4
No symbol "j" in current context.
#1  0x565555fb in main (argc=1, argv=0xffffd2c4) at fun.c:11
$1 = 5
(gdb) frame apply all -s p j
#1  0x565555fb in main (argc=1, argv=0xffffd2c4) at fun.c:11
$2 = 5
(gdb)
```

By default, '`frame apply`', prints the frame location information before the command output:

```
(gdb) frame apply all p $sp
#0  some_function (i=5) at fun.c:4
$4 = (void *) 0xffffd1e0
#1  0x565555fb in main (argc=1, argv=0xffffd2c4) at fun.c:11
$5 = (void *) 0xffffd1f0
(gdb)
```

If the flag `-q` is given, no frame information is printed:

```
(gdb) frame apply all -q p $sp
$12 = (void *) 0xffffd1e0
$13 = (void *) 0xffffd1f0
(gdb)
```

`faas` *command*

Shortcut for `frame apply all -s` *command*. Applies *command* on all frames, ignoring errors and empty output.

It can for example be used to print a local variable or a function argument without knowing the frame where this variable or argument is, using:

```
(gdb) faas p some_local_var_i_do_not_remember_where_it_is
```

The `faas` command accepts the same options as the `frame apply` command. See Section 8.5 [frame apply], page 114.

Note that the command `tfaas` *command* applies *command* on all frames of all threads. See See Section 4.10 [Threads], page 44.

8.6 Management of Frame Filters.

Frame filters are Python based utilities to manage and decorate the output of frames. See Section 23.3.2.9 [Frame Filter API], page 402, for further information.

Managing frame filters is performed by several commands available within GDB, detailed here.

`info frame-filter`

Print a list of installed frame filters from all dictionaries, showing their name, priority and enabled status.

`disable frame-filter` *filter-dictionary filter-name*

Disable a frame filter in the dictionary matching *filter-dictionary* and *filter-name*. The *filter-dictionary* may be `all`, `global`, `progspace`, or the name of

the object file where the frame filter dictionary resides. When `all` is specified, all frame filters across all dictionaries are disabled. The *filter-name* is the name of the frame filter and is used when `all` is not the option for *filter-dictionary*. A disabled frame-filter is not deleted, it may be enabled again later.

`enable frame-filter` *`filter-dictionary filter-name`*

Enable a frame filter in the dictionary matching *filter-dictionary* and *filter-name*. The *filter-dictionary* may be `all`, `global`, `progspace` or the name of the object file where the frame filter dictionary resides. When `all` is specified, all frame filters across all dictionaries are enabled. The *filter-name* is the name of the frame filter and is used when `all` is not the option for *filter-dictionary*.

Example:

```
(gdb) info frame-filter

global frame-filters:
  Priority  Enabled  Name
  1000      No       PrimaryFunctionFilter
  100       Yes      Reverse

progspace /build/test frame-filters:
  Priority  Enabled  Name
  100       Yes      ProgspaceFilter

objfile /build/test frame-filters:
  Priority  Enabled  Name
  999       Yes      BuildProgramFilter

(gdb) disable frame-filter /build/test BuildProgramFilter
(gdb) info frame-filter

global frame-filters:
  Priority  Enabled  Name
  1000      No       PrimaryFunctionFilter
  100       Yes      Reverse

progspace /build/test frame-filters:
  Priority  Enabled  Name
  100       Yes      ProgspaceFilter

objfile /build/test frame-filters:
  Priority  Enabled  Name
  999       No       BuildProgramFilter

(gdb) enable frame-filter global PrimaryFunctionFilter
(gdb) info frame-filter

global frame-filters:
  Priority  Enabled  Name
  1000      Yes      PrimaryFunctionFilter
  100       Yes      Reverse

progspace /build/test frame-filters:
  Priority  Enabled  Name
  100       Yes      ProgspaceFilter

objfile /build/test frame-filters:
```

```
  Priority  Enabled  Name
  999       No       BuildProgramFilter
```

`set frame-filter priority` *`filter-dictionary filter-name priority`*

Set the *priority* of a frame filter in the dictionary matching *filter-dictionary*, and the frame filter name matching *filter-name*. The *filter-dictionary* may be `global`, `progspace` or the name of the object file where the frame filter dictionary resides. The *priority* is an integer.

`show frame-filter priority` *`filter-dictionary filter-name`*

Show the *priority* of a frame filter in the dictionary matching *filter-dictionary*, and the frame filter name matching *filter-name*. The *filter-dictionary* may be `global`, `progspace` or the name of the object file where the frame filter dictionary resides.

Example:

```
(gdb) info frame-filter

global frame-filters:
  Priority  Enabled  Name
  1000      Yes      PrimaryFunctionFilter
  100       Yes      Reverse

progspace /build/test frame-filters:
  Priority  Enabled  Name
  100       Yes      ProgspaceFilter

objfile /build/test frame-filters:
  Priority  Enabled  Name
  999       No       BuildProgramFilter

(gdb) set frame-filter priority global Reverse 50
(gdb) info frame-filter

global frame-filters:
  Priority  Enabled  Name
  1000      Yes      PrimaryFunctionFilter
  50        Yes      Reverse

progspace /build/test frame-filters:
  Priority  Enabled  Name
  100       Yes      ProgspaceFilter

objfile /build/test frame-filters:
  Priority  Enabled  Name
  999       No       BuildProgramFilter
```

9 Examining Source Files

GDB can print parts of your program's source, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see Section 8.3 [Selecting a Frame], page 111), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source; see Chapter 26 [Using GDB under GNU Emacs], page 535.

9.1 Printing Source Lines

To print lines from a source file, use the `list` command (abbreviated `l`). By default, ten lines are printed. There are several ways to specify what part of the file you want to print; see Section 9.2 [Specify Location], page 120, for the full list.

Here are the forms of the `list` command most commonly used:

`list` *linenum*
: Print lines centered around line number *linenum* in the current source file.

`list` *function*
: Print lines centered around the beginning of function *function*.

`list`
: Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see Chapter 8 [Examining the Stack], page 107), this prints lines centered around that line.

`list -`
: Print lines just before the lines last printed.

By default, GDB prints ten source lines with any of these forms of the `list` command. You can change this using `set listsize`:

`set listsize` *count*
`set listsize unlimited`
: Make the `list` command display *count* source lines (unless the `list` argument explicitly specifies some other number). Setting *count* to `unlimited` or 0 means there's no limit.

`show listsize`
: Display the number of lines that `list` prints.

Repeating a `list` command with `RET` discards the argument, so it is equivalent to typing just `list`. This is more useful than listing the same lines again. An exception is made for an argument of '`-`'; that argument is preserved in repetition so that each repetition moves up in the source file.

In general, the `list` command expects you to supply zero, one or two *locations*. Locations specify source lines; there are several ways of writing them (see Section 9.2 [Specify Location], page 120), but the effect is always to specify some source line.

Here is a complete description of the possible arguments for `list`:

`list` *location*
: Print lines centered around the line specified by *location*.

`list` *first*`,`*last*
: Print lines from *first* to *last*. Both arguments are locations. When a `list` command has two locations, and the source file of the second location is omitted, this refers to the same source file as the first location.

`list ,`*last*
: Print lines ending with *last*.

`list` *first*`,`
: Print lines starting with *first*.

`list +`
: Print lines just after the lines last printed.

`list -`
: Print lines just before the lines last printed.

`list`
: As described in the preceding table.

9.2 Specifying a Location

Several GDB commands accept arguments that specify a location of your program's code. Since GDB is a source-level debugger, a location usually specifies some line in the source code. Locations may be specified using three different formats: linespec locations, explicit locations, or address locations.

9.2.1 Linespec Locations

A *linespec* is a colon-separated list of source location parameters such as file name, function name, etc. Here are all the different ways of specifying a linespec:

linenum
: Specifies the line number *linenum* of the current source file.

-offset

+offset
: Specifies the line *offset* lines before or after the *current line*. For the `list` command, the current line is the last one printed; for the breakpoint commands, this is the line at which execution stopped in the currently selected *stack frame* (see Section 8.1 [Frames], page 107, for a description of stack frames.) When used as the second of the two linespecs in a `list` command, this specifies the line *offset* lines up or down from the first linespec.

filename`:`*linenum*
: Specifies the line *linenum* in the source file *filename*. If *filename* is a relative file name, then it will match any source file name with the same trailing components. For example, if *filename* is '`gcc/expr.c`', then it will match source file name of `/build/trunk/gcc/expr.c`, but not `/build/trunk/libcpp/expr.c` or `/build/trunk/gcc/x-expr.c`.

function
: Specifies the line that begins the body of the function *function*. For example, in C, this is the line with the open brace.

  By default, in C++ and Ada, *function* is interpreted as specifying all functions named *function* in all scopes. For C++, this means in all namespaces and classes. For Ada, this means in all packages.

  For example, assuming a program with C++ symbols named `A::B::func` and `B::func`, both commands *`break func`* and *`break B::func`* set a breakpoint on both symbols.

Commands that accept a linespec let you override this with the `-qualified` option. For example, *`break -qualified func`* sets a breakpoint on a free-function named `func` ignoring any C++ class methods and namespace functions called `func`.

See Section 9.2.2 [Explicit Locations], page 121.

`function``:`*`label`*
Specifies the line where *label* appears in *function*.

`filename``:`*`function`*
Specifies the line that begins the body of the function *function* in the file *filename*. You only need the file name with a function name to avoid ambiguity when there are identically named functions in different source files.

`label` Specifies the line at which the label named *label* appears in the function corresponding to the currently selected stack frame. If there is no current selected stack frame (for instance, if the inferior is not running), then GDB will not search for a label.

`-pstap|-probe-stap` [*`objfile`*`:`[*`provider`*`:`]]*`name`*
The GNU/Linux tool `SystemTap` provides a way for applications to embed static probes. See Section 5.1.10 [Static Probe Points], page 77, for more information on finding and using static probes. This form of linespec specifies the location of such a static probe.

If *objfile* is given, only probes coming from that shared library or executable matching *objfile* as a regular expression are considered. If *provider* is given, then only probes from that provider are considered. If several probes match the spec, GDB will insert a breakpoint at each one of those probes.

9.2.2 Explicit Locations

Explicit locations allow the user to directly specify the source location's parameters using option-value pairs.

Explicit locations are useful when several functions, labels, or file names have the same name (base name for files) in the program's sources. In these cases, explicit locations point to the source line you meant more accurately and unambiguously. Also, using explicit locations might be faster in large programs.

For example, the linespec '`foo:bar`' may refer to a function `bar` defined in the file named `foo` or the label `bar` in a function named `foo`. GDB must search either the file system or the symbol table to know.

The list of valid explicit location options is summarized in the following table:

`-source` *`filename`*
The value specifies the source file name. To differentiate between files with the same base name, prepend as many directories as is necessary to uniquely identify the desired file, e.g., `foo/bar/baz.c`. Otherwise GDB will use the first file it finds with the given base name. This option requires the use of either `-function` or `-line`.

`-function` *function*

The value specifies the name of a function. Operations on function locations unmodified by other options (such as `-label` or `-line`) refer to the line that begins the body of the function. In C, for example, this is the line with the open brace.

By default, in C++ and Ada, *function* is interpreted as specifying all functions named *function* in all scopes. For C++, this means in all namespaces and classes. For Ada, this means in all packages.

For example, assuming a program with C++ symbols named `A::B::func` and `B::func`, both commands *`break -function func`* and *`break -function B::func`* set a breakpoint on both symbols.

You can use the *`-qualified`* flag to override this (see below).

`-qualified`

This flag makes GDB interpret a function name specified with *`-function`* as a complete fully-qualified name.

For example, assuming a C++ program with symbols named `A::B::func` and `B::func`, the *`break -qualified -function B::func`* command sets a breakpoint on `B::func`, only.

(Note: the *`-qualified`* option can precede a linespec as well (see Section 9.2.1 [Linespec Locations], page 120), so the particular example above could be simplified as *`break -qualified B::func`*.)

`-label` *label*

The value specifies the name of a label. When the function name is not specified, the label is searched in the function of the currently selected stack frame.

`-line` *number*

The value specifies a line offset for the location. The offset may either be absolute (`-line 3`) or relative (`-line +3`), depending on the command. When specified without any other options, the line offset is relative to the current line.

Explicit location options may be abbreviated by omitting any non-unique trailing characters from the option name, e.g., *`break -s main.c -li 3`*.

9.2.3 Address Locations

Address locations indicate a specific program address. They have the generalized form **address*.

For line-oriented commands, such as `list` and `edit`, this specifies a source line that contains *address*. For `break` and other breakpoint-oriented commands, this can be used to set breakpoints in parts of your program which do not have debugging information or source files.

Here *address* may be any expression valid in the current working language (see Chapter 15 [Languages], page 217) that specifies a code address. In addition, as a convenience, GDB extends the semantics of expressions used in locations to cover several situations that frequently occur during debugging. Here are the various forms of *address*:

`expression`

Any expression valid in the current working language.

funcaddr An address of a function or procedure derived from its name. In C, C++, Objective-C, Fortran, minimal, and assembly, this is simply the function's name *function* (and actually a special case of a valid expression). In Pascal and Modula-2, this is `&function`. In Ada, this is `function'Address` (although the Pascal form also works).

This form specifies the address of the function's first instruction, before the stack frame and arguments have been set up.

'filename':funcaddr
Like *funcaddr* above, but also specifies the name of the source file explicitly. This is useful if the name of the function does not specify the function unambiguously, e.g., if there are several functions with identical names in different source files.

9.3 Editing Source Files

To edit the lines in a source file, use the `edit` command. The editing program of your choice is invoked with the current line set to the active line in the program. Alternatively, there are several ways to specify what part of the file you want to print if you want to see other parts of the program:

`edit` *location*
Edit the source file specified by `location`. Editing starts at that *location*, e.g., at the specified source line of the specified file. See Section 9.2 [Specify Location], page 120, for all the possible forms of the *location* argument; here are the forms of the `edit` command most commonly used:

`edit` *number*
Edit the current source file with *number* as the active line number.

`edit` *function*
Edit the file containing *function* at the beginning of its definition.

9.3.1 Choosing your Editor

You can customize GDB to use any editor you want[1]. By default, it is `/bin/ex`, but you can change this by setting the environment variable `EDITOR` before using GDB. For example, to configure GDB to use the `vi` editor, you could use these commands with the `sh` shell:

```
EDITOR=/usr/bin/vi
export EDITOR
gdb ...
```

or in the `csh` shell,

```
setenv EDITOR /usr/bin/vi
gdb ...
```

[1] The only restriction is that your editor (say `ex`), recognizes the following command-line syntax:

```
ex +number file
```

The optional numeric value *+number* specifies the number of the line in the file where to start editing.

9.4 Searching Source Files

There are two commands for searching through the current source file for a regular expression.

`forward-search` *regexp*
`search` *regexp*
: The command '`forward-search` *regexp*' checks each line, starting with the one following the last line listed, for a match for *regexp*. It lists the line that is found. You can use the synonym '`search` *regexp*' or abbreviate the command name as `fo`.

`reverse-search` *regexp*
: The command '`reverse-search` *regexp*' checks each line, starting with the one before the last line listed and going backward, for a match for *regexp*. It lists the line that is found. You can abbreviate this command as `rev`.

9.5 Specifying Source Directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, just the names. Even when they do, the directories could be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the *source path*. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name.

For example, suppose an executable references the file `/usr/src/foo-1.0/lib/foo.c`, does not record a compilation directory, and the *source path* is `/mnt/cross`. GDB would look for the source file in the following locations:

1. `/usr/src/foo-1.0/lib/foo.c`
2. `/mnt/cross/usr/src/foo-1.0/lib/foo.c`
3. `/mnt/cross/foo.c`

If the source file is not present at any of the above locations then an error is printed. GDB does not look up the parts of the source file name, such as `/mnt/cross/src/foo-1.0/lib/foo.c`. Likewise, the subdirectories of the source path are not searched: if the source path is `/mnt/cross`, and the binary refers to `foo.c`, GDB would not find it under `/mnt/cross/usr/src/foo-1.0/lib`.

Plain file names, relative file names with leading directories, file names containing dots, etc. are all treated as described above, except that non-absolute file names are not looked up literally. If the *source path* is `/mnt/cross`, the source file is recorded as `../lib/foo.c`, and no compilation directory is recorded, then GDB will search in the following locations:

1. `/mnt/cross/../lib/foo.c`
2. `/mnt/cross/foo.c`

The *source path* will always include two special entries '`$cdir`' and '`$cwd`', these refer to the compilation directory (if one is recorded) and the current working directory respectively.

'`$cdir`' causes GDB to search within the compilation directory, if one is recorded in the debug information. If no compilation directory is recorded in the debug information then '`$cdir`' is ignored.

'`$cwd`' is not the same as '`.`'—the former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

If a compilation directory is recorded in the debug information, and GDB has not found the source file after the first search using *source path*, then GDB will combine the compilation directory and the filename, and then search for the source file again using the *source path*.

For example, if the executable records the source file as `/usr/src/foo-1.0/lib/foo.c`, the compilation directory is recorded as `/project/build`, and the *source path* is `/mnt/cross:$cdir:$cwd` while the current working directory of the GDB session is `/home/user`, then GDB will search for the source file in the following locations:

1. `/usr/src/foo-1.0/lib/foo.c`
2. `/mnt/cross/usr/src/foo-1.0/lib/foo.c`
3. `/project/build/usr/src/foo-1.0/lib/foo.c`
4. `/home/user/usr/src/foo-1.0/lib/foo.c`
5. `/mnt/cross/project/build/usr/src/foo-1.0/lib/foo.c`
6. `/project/build/project/build/usr/src/foo-1.0/lib/foo.c`
7. `/home/user/project/build/usr/src/foo-1.0/lib/foo.c`
8. `/mnt/cross/foo.c`
9. `/project/build/foo.c`
10. `/home/user/foo.c`

If the file name in the previous example had been recorded in the executable as a relative path rather than an absolute path, then the first look up would not have occurred, but all of the remaining steps would be similar.

When searching for source files on MS-DOS and MS-Windows, where absolute paths start with a drive letter (e.g. `C:/project/foo.c`), GDB will remove the drive letter from the file name before appending it to a search directory from *source path*; for instance if the executable references the source file `C:/project/foo.c` and *source path* is set to `D:/mnt/cross`, then GDB will search in the following locations for the source file:

1. `C:/project/foo.c`
2. `D:/mnt/cross/project/foo.c`
3. `D:/mnt/cross/foo.c`

Note that the executable search path is *not* used to locate the source files.

Whenever you reset or rearrange the source path, GDB clears out any information it has cached about where source files are found and where each line is in the file.

When you start GDB, its source path includes only '`$cdir`' and '`$cwd`', in that order. To add other directories, use the `directory` command.

The search path is used to find both program source files and GDB script files (read using the '`-command`' option and '`source`' command).

In addition to the source path, GDB provides a set of commands that manage a list of source path substitution rules. A *substitution rule* specifies how to rewrite source directories stored in the program's debug information in case the sources were moved to a different

directory between compilation and debugging. A rule is made of two strings, the first specifying what needs to be rewritten in the path, and the second specifying how it should be rewritten. In [set substitute-path], page 127, we name these two parts *from* and *to* respectively. GDB does a simple string replacement of *from* with *to* at the start of the directory part of the source file name, and uses that result instead of the original file name to look up the sources.

Using the previous example, suppose the `foo-1.0` tree has been moved from `/usr/src` to `/mnt/cross`, then you can tell GDB to replace `/usr/src` in all source path names with `/mnt/cross`. The first lookup will then be `/mnt/cross/foo-1.0/lib/foo.c` in place of the original location of `/usr/src/foo-1.0/lib/foo.c`. To define a source path substitution rule, use the `set substitute-path` command (see [set substitute-path], page 127).

To avoid unexpected substitution results, a rule is applied only if the *from* part of the directory name ends at a directory separator. For instance, a rule substituting `/usr/source` into `/mnt/cross` will be applied to `/usr/source/foo-1.0` but not to `/usr/sourceware/foo-2.0`. And because the substitution is applied only at the beginning of the directory name, this rule will not be applied to `/root/usr/source/baz.c` either.

In many cases, you can achieve the same result using the `directory` command. However, `set substitute-path` can be more efficient in the case where the sources are organized in a complex tree with multiple subdirectories. With the `directory` command, you need to add each subdirectory of your project. If you moved the entire tree while preserving its internal organization, then `set substitute-path` allows you to direct the debugger to all the sources with one single command.

`set substitute-path` is also more than just a shortcut command. The source path is only used if the file at the original location no longer exists. On the other hand, `set substitute-path` modifies the debugger behavior to look at the rewritten location instead. So, if for any reason a source file that is not relevant to your executable is located at the original location, a substitution rule is the only method available to point GDB at the new location.

You can configure a default source path substitution rule by configuring GDB with the '`--with-relocated-sources=`*dir*' option. The *dir* should be the name of a directory under GDB's configured prefix (set with '`--prefix`' or '`--exec-prefix`'), and directory names in debug information under *dir* will be adjusted automatically if the installed GDB is moved to a new location. This is useful if GDB, libraries or executables with debug information and corresponding source code are being moved together.

`directory` *dirname* `...`
`dir` *dirname* `...`

Add directory *dirname* to the front of the source path. Several directory names may be given to this command, separated by '`:`' ('`;`' on MS-DOS and MS-Windows, where '`:`' usually appears as part of absolute file names) or white-space. You may specify a directory that is already in the source path; this moves it forward, so GDB searches it sooner.

The special strings '`$cdir`' (to refer to the compilation directory, if one is recorded), and '`$cwd`' (to refer to the current working directory) can also be included in the list of directories *dirname*. Though these will already be in the source path they will be moved forward in the list so GDB searches them sooner.

`directory`
: Reset the source path to its default value (‘`$cdir:$cwd`’ on Unix systems). This requires confirmation.

`set directories` *path-list*
: Set the source path to *path-list*. ‘`$cdir:$cwd`’ are added if missing.

`show directories`
: Print the source path: show which directories it contains.

`set substitute-path` *from to*
: Define a source path substitution rule, and add it at the end of the current list of existing substitution rules. If a rule with the same *from* was already defined, then the old rule is also deleted.

  For example, if the file `/foo/bar/baz.c` was moved to `/mnt/cross/baz.c`, then the command

  ```
  (gdb) set substitute-path /foo/bar /mnt/cross
  ```

  will tell GDB to replace ‘`/foo/bar`’ with ‘`/mnt/cross`’, which will allow GDB to find the file `baz.c` even though it was moved.

  In the case when more than one substitution rule have been defined, the rules are evaluated one by one in the order where they have been defined. The first one matching, if any, is selected to perform the substitution.

  For instance, if we had entered the following commands:

  ```
  (gdb) set substitute-path /usr/src/include /mnt/include
  (gdb) set substitute-path /usr/src /mnt/src
  ```

  GDB would then rewrite `/usr/src/include/defs.h` into `/mnt/include/defs.h` by using the first rule. However, it would use the second rule to rewrite `/usr/src/lib/foo.c` into `/mnt/src/lib/foo.c`.

`unset substitute-path [path]`
: If a path is specified, search the current list of substitution rules for a rule that would rewrite that path. Delete that rule if found. A warning is emitted by the debugger if no rule could be found.

  If no path is specified, then all substitution rules are deleted.

`show substitute-path [path]`
: If a path is specified, then print the source path substitution rule which would rewrite that path, if any.

  If no path is specified, then print all existing source path substitution rules.

If your source path is cluttered with directories that are no longer of interest, GDB may sometimes cause confusion by finding the wrong versions of source. You can correct the situation as follows:

1. Use `directory` with no argument to reset the source path to its default value.
2. Use `directory` with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

9.6 Source and Machine Code

You can use the command `info line` to map source lines to program addresses (and vice versa), and the command `disassemble` to display a range of addresses as machine instructions. You can use the command `set disassemble-next-line` to set whether to disassemble next source line when execution stops. When run under GNU Emacs mode, the `info line` command causes the arrow to point to the line specified. Also, `info line` prints addresses in symbolic form as well as hex.

`info line`
`info line` *location*

Print the starting and ending addresses of the compiled code for source line *location*. You can specify source lines in any of the ways documented in Section 9.2 [Specify Location], page 120. With no *location* information about the current source line is printed.

For example, we can use `info line` to discover the location of the object code for the first line of function `m4_changequote`:

```
(gdb) info line m4_changequote
Line 895 of "builtin.c" starts at pc 0x634c <m4_changequote> and \
        ends at 0x6350 <m4_changequote+4>.
```

We can also inquire (using `*addr` as the form for *location*) what source line covers a particular address:

```
(gdb) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 <m4_changequote+152> and \
        ends at 0x6404 <m4_changequote+184>.
```

After `info line`, the default address for the `x` command is changed to the starting address of the line, so that '`x/i`' is sufficient to begin examining the machine code (see Section 10.6 [Examining Memory], page 143). Also, this address is saved as the value of the convenience variable `$_` (see Section 10.12 [Convenience Variables], page 162).

After `info line`, using `info line` again without specifying a location will display information about the next source line.

`disassemble`
`disassemble /m`
`disassemble /s`
`disassemble /r`

This specialized command dumps a range of memory as machine instructions. It can also print mixed source+disassembly by specifying the `/m` or `/s` modifier and print the raw instructions in hex as well as in symbolic form by specifying the `/r` modifier. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; GDB dumps the function surrounding this value. When two arguments are given, they should be separated by a comma, possibly surrounded by whitespace. The arguments specify a range of addresses to dump, in one of two forms:

start`,`*end*

the addresses from *start* (inclusive) to *end* (exclusive)

start,+*length*
: the addresses from *start* (inclusive) to *start*+*length* (exclusive).

When 2 arguments are specified, the name of the function is also printed (since there could be several functions in the given range).

The argument(s) can be any expression yielding a numeric value, such as '0x32c4', '&main+10' or '$pc - 8'.

If the range of memory being disassembled contains current program counter, the instruction at that location is shown with a => marker.

The following example shows the disassembly of a range of addresses of HP PA-RISC 2.0 code:

```
(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.
```

Here is an example showing mixed source+assembly for Intel x86 with /m or /s, when the program is stopped just after function prologue in a non-optimized function with no inline code.

```
(gdb) disas /m main
Dump of assembler code for function main:
5       {
   0x08048330 <+0>:    push   %ebp
   0x08048331 <+1>:    mov    %esp,%ebp
   0x08048333 <+3>:    sub    $0x8,%esp
   0x08048336 <+6>:    and    $0xfffffff0,%esp
   0x08048339 <+9>:    sub    $0x10,%esp

6         printf ("Hello.\n");
=> 0x0804833c <+12>:   movl   $0x8048440,(%esp)
   0x08048343 <+19>:   call   0x8048284 <puts@plt>

7         return 0;
8       }
   0x08048348 <+24>:   mov    $0x0,%eax
   0x0804834d <+29>:   leave
   0x0804834e <+30>:   ret

End of assembler dump.
```

The /m option is deprecated as its output is not useful when there is either inlined code or re-ordered code. The /s option is the preferred choice. Here is an example for AMD x86-64 showing the difference between /m output and /s output. This example has one inline function defined in a header file, and the code is compiled with '-O2' optimization. Note how the /m output is missing the disassembly of several instructions that are present in the /s output.

foo.h:

```
int
foo (int a)
{
  if (a < 0)
    return a * 2;
  if (a == 0)
    return 1;
  return a + 10;
}
```

foo.c:

```
#include "foo.h"
volatile int x, y;
int
main ()
{
  x = foo (y);
  return 0;
}
```

```
(gdb) disas /m main
Dump of assembler code for function main:
5 {

6   x = foo (y);
   0x0000000000400400 <+0>:  mov    0x200c2e(%rip),%eax # 0x601034 <y>
   0x0000000000400417 <+23>:  mov    %eax,0x200c13(%rip) # 0x601030 <x>

7   return 0;
8 }
   0x000000000040041d <+29>:  xor    %eax,%eax
   0x000000000040041f <+31>:  retq
   0x0000000000400420 <+32>:  add    %eax,%eax
   0x0000000000400422 <+34>:  jmp    0x400417 <main+23>

End of assembler dump.
(gdb) disas /s main
Dump of assembler code for function main:
foo.c:
5 {
6   x = foo (y);
   0x0000000000400400 <+0>:  mov    0x200c2e(%rip),%eax # 0x601034 <y>

foo.h:
4   if (a < 0)
   0x0000000000400406 <+6>:  test   %eax,%eax
   0x0000000000400408 <+8>:  js     0x400420 <main+32>

6   if (a == 0)
7     return 1;
8   return a + 10;
   0x000000000040040a <+10>:  lea    0xa(%rax),%edx
   0x000000000040040d <+13>:  test   %eax,%eax
   0x000000000040040f <+15>:  mov    $0x1,%eax
   0x0000000000400414 <+20>:  cmovne %edx,%eax

foo.c:
6   x = foo (y);
```

```
   0x0000000000400417 <+23>:  mov    %eax,0x200c13(%rip) # 0x601030 <x>

7   return 0;
8 }
   0x000000000040041d <+29>:  xor    %eax,%eax
   0x000000000040041f <+31>:  retq

foo.h:
5     return a * 2;
   0x0000000000400420 <+32>:  add    %eax,%eax
   0x0000000000400422 <+34>:  jmp    0x400417 <main+23>
End of assembler dump.
```

Here is another example showing raw instructions in hex for AMD x86-64,

```
(gdb) disas /r 0x400281,+10
Dump of assembler code from 0x400281 to 0x40028b:
   0x0000000000400281:  38 36  cmp    %dh,(%rsi)
   0x0000000000400283:  2d 36 34 2e 73 sub    $0x732e3436,%eax
   0x0000000000400288:  6f     outsl  %ds:(%rsi),(%dx)
   0x0000000000400289:  2e 32 00       xor    %cs:(%rax),%al
End of assembler dump.
```

Addresses cannot be specified as a location (see Section 9.2 [Specify Location], page 120). So, for example, if you want to disassemble function `bar` in file `foo.c`, you must type '`disassemble 'foo.c'::bar`' and not '`disassemble foo.c:bar`'.

Some architectures have more than one commonly-used set of instruction mnemonics or other syntax.

For programs that were dynamically linked and use shared libraries, instructions that call functions or branch to locations in the shared libraries might show a seemingly bogus location—it's actually a location of the relocation table. On some architectures, GDB might be able to resolve these to actual function names.

`set disassembler-options` *`option1`*`[,`*`option2`*`...]`

This command controls the passing of target specific information to the disassembler. For a list of valid options, please refer to the `-M`/`--disassembler-options` section of the '`objdump`' manual and/or the output of *`objdump --help`* (see Section "objdump" in *The GNU Binary Utilities*). The default value is the empty string.

If it is necessary to specify more than one disassembler option, then multiple options can be placed together into a comma separated list. Currently this command is only supported on targets ARC, ARM, MIPS, PowerPC and S/390.

`show disassembler-options`

Show the current setting of the disassembler options.

`set disassembly-flavor` *`instruction-set`*

Select the instruction set to use when disassembling the program via the `disassemble` or `x/i` commands.

Currently this command is only defined for the Intel x86 family. You can set *instruction-set* to either `intel` or `att`. The default is `att`, the AT&T flavor used by default by Unix assemblers for x86-based targets.

`show disassembly-flavor`

Show the current setting of the disassembly flavor.

`set disassemble-next-line`
`show disassemble-next-line`

Control whether or not GDB will disassemble the next source line or instruction when execution stops. If ON, GDB will display disassembly of the next source line when execution of the program being debugged stops. This is *in addition* to displaying the source line itself, which GDB always does if possible. If the next source line cannot be displayed for some reason (e.g., if GDB cannot find the source file, or there's no line info in the debug info), GDB will display disassembly of the next *instruction* instead of showing the next source line. If AUTO, GDB will display disassembly of next instruction only if the source line cannot be displayed. This setting causes GDB to display some feedback when you step through a function with no line info or whose source file is unavailable. The default is OFF, which means never display the disassembly of the next line or instruction.

9.7 Disable Reading Source Code

In some cases it can be desirable to prevent GDB from accessing source code files. One case where this might be desirable is if the source code files are located over a slow network connection.

The following command can be used to control whether GDB should access source code files or not:

`set source open` [`on`|`off`]
`show source open`

When this option is `on`, which is the default, GDB will access source code files when needed, for example to print source lines when GDB stops, or in response to the `list` command.

When this option is `off`, GDB will not access source code files.

10 Examining Data

The usual way to examine data in your program is with the `print` command (abbreviated `p`), or its synonym `inspect`. It evaluates and prints the value of an expression of the language your program is written in (see Chapter 15 [Using GDB with Different Languages], page 217). It may also print the expression using a Python-based pretty-printer (see Section 10.10 [Pretty Printing], page 159).

`print [[`*options*`] --] `*expr*
`print [[`*options*`] --] /`*f* *expr*

expr is an expression (in the source language). By default the value of *expr* is printed in a format appropriate to its data type; you can choose a different format by specifying '`/`*f*', where *f* is a letter specifying the format; see Section 10.5 [Output Formats], page 142.

The `print` command supports a number of options that allow overriding relevant global print settings as set by `set print` subcommands:

`-address [on|off]`
: Set printing of addresses. Related setting: [set print address], page 149.

`-array [on|off]`
: Pretty formatting of arrays. Related setting: [set print array], page 150.

`-array-indexes [on|off]`
: Set printing of array indexes. Related setting: [set print array-indexes], page 150.

`-elements` *number-of-elements*`|unlimited`
: Set limit on string chars or array elements to print. The value `unlimited` causes there to be no limit. Related setting: [set print elements], page 151.

`-max-depth` *depth*`|unlimited`
: Set the threshold after which nested structures are replaced with ellipsis. Related setting: [set print max-depth], page 154.

`-memory-tag-violations [on|off]`
: Set printing of additional information about memory tag violations. See [set print memory-tag-violations], page 155.

`-null-stop [on|off]`
: Set printing of char arrays to stop at first null char. Related setting: [set print null-stop], page 155.

`-object [on|off]`
: Set printing C++ virtual function tables. Related setting: [set print object], page 158.

`-pretty [on|off]`
: Set pretty formatting of structures. Related setting: [set print pretty], page 156.

`-raw-values [on|off]`
: Set whether to print values in raw form, bypassing any pretty-printers for that value. Related setting: [set print raw-values], page 156.

`-repeats` *`number-of-repeats`*`|unlimited`
: Set threshold for repeated print elements. `unlimited` causes all elements to be individually printed. Related setting: [set print repeats], page 154.

`-static-members [on|off]`
: Set printing C++ static members. Related setting: [set print static-members], page 158.

`-symbol [on|off]`
: Set printing of symbol names when printing pointers. Related setting: [set print symbol], page 150.

`-union [on|off]`
: Set printing of unions interior to structures. Related setting: [set print union], page 156.

`-vtbl [on|off]`
: Set printing of C++ virtual function tables. Related setting: [set print vtbl], page 158.

Because the `print` command accepts arbitrary expressions which may look like options (including abbreviations), if you specify any command option, then you must use a double dash (`--`) to mark the end of option processing.

For example, this prints the value of the `-p` expression:

```
(gdb) print -p
```

While this repeats the last value in the value history (see below) with the `-pretty` option in effect:

```
(gdb) print -p --
```

Here is an example including both on option and an expression:

```
(gdb) print -pretty -- *myptr
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

`print [`*`options`*`]`
`print [`*`options`*`] /f`
: If you omit *expr*, GDB displays the last value again (from the *value history*; see Section 10.11 [Value History], page 161). This allows you to conveniently inspect the same value in an alternative format.

If the architecture supports memory tagging, the `print` command will display pointer/memory tag mismatches if what is being printed is a pointer or reference type. See Section 10.7 [Memory Tagging], page 146.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. See Section 10.6 [Examining Memory], page 143.

If you are interested in information about types, or about how the fields of a struct or a class are declared, use the `ptype` *exp* command rather than `print`. See Chapter 16 [Examining the Symbol Table], page 251.

Another way of examining values of expressions and type information is through the Python extension command `explore` (available only if the GDB build is configured with `--with-python`). It offers an interactive way to start at the highest level (or, the most abstract level) of the data type of an expression (or, the data type itself) and explore all the way down to leaf scalar values/fields embedded in the higher level data types.

`explore` *arg*

> *arg* is either an expression (in the source language), or a type visible in the current context of the program being debugged.

The working of the `explore` command can be illustrated with an example. If a data type `struct ComplexStruct` is defined in your C program as

```
struct SimpleStruct
{
  int i;
  double d;
};

struct ComplexStruct
{
  struct SimpleStruct *ss_p;
  int arr[10];
};
```

followed by variable declarations as

```
struct SimpleStruct ss = { 10, 1.11 };
struct ComplexStruct cs = { &ss, { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } };
```

then, the value of the variable `cs` can be explored using the `explore` command as follows.

```
(gdb) explore cs
The value of ‘cs’ is a struct/class of type ‘struct ComplexStruct’ with
the following fields:

  ss_p = <Enter 0 to explore this field of type ‘struct SimpleStruct *’>
   arr = <Enter 1 to explore this field of type ‘int [10]’>

Enter the field number of choice:
```

Since the fields of `cs` are not scalar values, you are being prompted to chose the field you want to explore. Let's say you choose the field `ss_p` by entering `0`. Then, since this field is a pointer, you will be asked if it is pointing to a single value. From the declaration of `cs` above, it is indeed pointing to a single value, hence you enter `y`. If you enter `n`, then you will be asked if it were pointing to an array of values, in which case this field will be explored as if it were an array.

```
‘cs.ss_p’ is a pointer to a value of type ‘struct SimpleStruct’
```

```
Continue exploring it as a pointer to a single value [y/n]: y
The value of '*(cs.ss_p)' is a struct/class of type 'struct
SimpleStruct' with the following fields:

  i = 10 .. (Value of type 'int')
  d = 1.1100000000000001 .. (Value of type 'double')

Press enter to return to parent value:
```

If the field `arr` of `cs` was chosen for exploration by entering `1` earlier, then since it is as array, you will be prompted to enter the index of the element in the array that you want to explore.

```
'cs.arr' is an array of 'int'.
Enter the index of the element you want to explore in 'cs.arr': 5

'(cs.arr)[5]' is a scalar value of type 'int'.

(cs.arr)[5] = 4

Press enter to return to parent value:
```

In general, at any stage of exploration, you can go deeper towards the leaf values by responding to the prompts appropriately, or hit the return key to return to the enclosing data structure (the *higher* level data structure).

Similar to exploring values, you can use the `explore` command to explore types. Instead of specifying a value (which is typically a variable name or an expression valid in the current context of the program being debugged), you specify a type name. If you consider the same example as above, your can explore the type `struct ComplexStruct` by passing the argument `struct ComplexStruct` to the `explore` command.

```
(gdb) explore struct ComplexStruct
```

By responding to the prompts appropriately in the subsequent interactive session, you can explore the type `struct ComplexStruct` in a manner similar to how the value `cs` was explored in the above example.

The `explore` command also has two sub-commands, `explore value` and `explore type`. The former sub-command is a way to explicitly specify that value exploration of the argument is being invoked, while the latter is a way to explicitly specify that type exploration of the argument is being invoked.

`explore value` *expr*
: This sub-command of `explore` explores the value of the expression *expr* (if *expr* is an expression valid in the current context of the program being debugged). The behavior of this command is identical to that of the behavior of the `explore` command being passed the argument *expr*.

`explore type` *arg*
: This sub-command of `explore` explores the type of *arg* (if *arg* is a type visible in the current context of program being debugged), or the type of the value/expression *arg* (if *arg* is an expression valid in the current context of the program being debugged). If *arg* is a type, then the behavior of this command is identical to that of the `explore` command being passed the argument *arg*. If *arg* is an expression, then the behavior of this command will be identical to that of the `explore` command being passed the type of *arg* as the argument.

10.1 Expressions

`print` and many other GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is valid in an expression in GDB. This includes conditional expressions, function calls, casts, and string constants. It also includes preprocessor macros, if you compiled your program to include this information; see Section 4.1 [Compilation], page 31.

GDB supports array constants in expressions input by the user. The syntax is {*element*, *element*...}. For example, you can use the command `print {1, 2, 3}` to create an array of three integers. If you pass an array to a function or assign it to a program variable, GDB copies the array to memory that is `malloc`ed in the target program.

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See Chapter 15 [Using GDB with Different Languages], page 217, for information on how to use expressions in other languages.

In this section, we discuss operators that you can use in GDB expressions regardless of your programming language.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into a pointer in order to examine a structure at that address in memory.

GDB supports these operators, in addition to those common to programming languages:

`@` '`@`' is a binary operator for treating parts of memory as arrays. See Section 10.4 [Artificial Arrays], page 141, for more information.

`::` '`::`' allows you to specify a variable in terms of the file or function where it is defined. See Section 10.3 [Program Variables], page 138.

`{`*type*`}` *addr*
: Refers to an object of type *type* stored at address *addr* in memory. The address *addr* may be any expression whose value is an integer or pointer (but parentheses are required around binary operators, just as in a cast). This construct is allowed regardless of what kind of data is normally supposed to reside at *addr*.

10.2 Ambiguous Expressions

Expressions can sometimes contain some ambiguous elements. For instance, some programming languages (notably Ada, C++ and Objective-C) permit a single function name to be defined several times, for application in different contexts. This is called *overloading*. Another example involving Ada is generics. A *generic package* is similar to C++ templates and is typically instantiated several times, resulting in the same function name being defined in different contexts.

In some cases and depending on the language, it is possible to adjust the expression to remove the ambiguity. For instance in C++, you can specify the signature of the function you want to break on, as in *`break function(types)`*. In Ada, using the fully qualified name of your function often makes the expression unambiguous as well.

When an ambiguity that needs to be resolved is detected, the debugger has the capability to display a menu of numbered choices for each possibility, and then waits for the selection with the prompt '`>`'. The first option is always '`[0] cancel`', and typing *`0 RET`* aborts the current command. If the command in which the expression was used allows more than one

choice to be selected, the next option in the menu is '`[1] all`', and typing *`1 RET`* selects all possible choices.

For example, the following session excerpt shows an attempt to set a breakpoint at the overloaded symbol `String::after`. We choose three particular definitions of that function name:

```
(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)
```

`set multiple-symbols` *mode*

This option allows you to adjust the debugger behavior when an expression is ambiguous.

By default, *mode* is set to `all`. If the command with which the expression is used allows more than one choice, then GDB automatically selects all possible choices. For instance, inserting a breakpoint on a function using an ambiguous name results in a breakpoint inserted on each possible match. However, if a unique choice must be made, then GDB uses the menu to help you disambiguate the expression. For instance, printing the address of an overloaded function will result in the use of the menu.

When *mode* is set to `ask`, the debugger always uses the menu when an ambiguity is detected.

Finally, when *mode* is set to `cancel`, the debugger reports an error due to the ambiguity and the command is aborted.

`show multiple-symbols`

Show the current value of the `multiple-symbols` setting.

10.3 Program Variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see Section 8.3 [Selecting a Frame], page 111); they must be either:

- global (or file-static)

or

- visible according to the scope rules of the programming language from the point of execution in that frame

This means that in the function

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

you can examine and use the variable `a` whenever your program is executing within the function `foo`, but you can only use or examine the variable `b` while your program is executing inside the block where `b` is declared.

There is an exception: you can refer to a variable or function whose scope is a single source file even if the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a static variable in a particular function or file by using the colon-colon (`::`) notation:

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

Here *file* or *function* is the name of the context for the static *variable*. In the case of file names, you can use quotes to make sure GDB parses the file name as a single word—for example, to print a global value of `x` defined in `f2.c`:

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

The `::` notation is normally used for referring to static variables, since you typically disambiguate uses of local variables in functions by selecting the appropriate frame and using the simple name of the variable. However, you may also use this notation to refer to local variables in frames enclosing the selected frame:

```
void
foo (int a)
{
  if (a < 10)
    bar (a);
  else
    process (a);    /* Stop here */
}

int
bar (int a)
{
  foo (a + 5);
}
```

For example, if there is a breakpoint at the commented line, here is what you might see when the program stops after executing the call `bar(0)`:

```
(gdb) p a
$1 = 10
(gdb) p bar::a
$2 = 5
(gdb) up 2
#2  0x080483d0 in foo (a=5) at foobar.c:12
(gdb) p a
```

```
$3 = 5
(gdb) p bar::a
$4 = 0
```

These uses of '`::`' are very rarely in conflict with the very similar use of the same notation in C++. When they are in conflict, the C++ meaning takes precedence; however, this can be overridden by quoting the file or function name with single quotes.

For example, suppose the program is stopped in a method of a class that has a field named `includefile`, and there is also an include file named `includefile` that defines a variable, `some_global`.

```
(gdb) p includefile
$1 = 23
(gdb) p includefile::some_global
A syntax error in expression, near ‘’.
(gdb) p 'includefile'::some_global
$2 = 27
```

Warning: Occasionally, a local variable may appear to have the wrong value at certain points in a function—just after entry to a new scope, and just before exit.

You may see this problem when you are stepping by machine instructions. This is because, on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually also takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

This may also happen when the compiler does significant optimizations. To be sure of always seeing accurate values, turn off all optimization when compiling.

Another possible effect of compiler optimizations is to optimize unused variables out of existence, or assign variables to registers (as opposed to memory addresses). Depending on the support for such cases offered by the debug info format used by the compiler, GDB might not be able to display values for such local variables. If that happens, GDB will print a message like this:

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

To solve such problems, either recompile without optimizations, or use a different debug info format, if the compiler supports several such formats. See Section 4.1 [Compilation], page 31, for more information on choosing compiler options. See Section 15.4.1 [C and C++], page 221, for more information about debug info formats that are best suited to C++ programs.

If you ask to print an object whose contents are unknown to GDB, e.g., because its data type is not completely specified by the debug information, GDB will say '`<incomplete type>`'. See Chapter 16 [Symbols], page 251, for more about this.

If you try to examine or use the value of a (global) variable for which GDB has no type information, e.g., because the program includes no debug information, GDB displays an error message. See Chapter 16 [Symbols], page 251, for more about unknown types. If you cast the variable to its declared type, GDB gets the variable's value using the cast-to type as the variable's type. For example, in a C program:

```
(gdb) p var
```

```
'var' has unknown type; cast it to its declared type
(gdb) p (float) var
$1 = 3.14
```

If you append *@entry* string to a function parameter name you get its value at the time the function got called. If the value is not available an error message is printed. Entry values are available only with some compilers. Entry values are normally also printed at the function parameter list according to [set print entry-values], page 152.

```
Breakpoint 1, d (i=30) at gdb.base/entry-value.c:29
29   i++;
(gdb) next
30   e (i);
(gdb) print i
$1 = 31
(gdb) print i@entry
$2 = 30
```

Strings are identified as arrays of `char` values without specified signedness. Arrays of either `signed char` or `unsigned char` get printed as arrays of 1 byte sized integers. `-fsigned-char` or `-funsigned-char` GCC options have no effect as GDB defines literal string type `"char"` as `char` without a sign. For program code

```
char var0[] = "A";
signed char var1[] = "A";
```

You get during debugging

```
(gdb) print var0
$1 = "A"
(gdb) print var1
$2 = {65 'A', 0 '\0'}
```

10.4 Artificial Arrays

It is often useful to print out several successive objects of the same type in memory; a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

You can do this by referring to a contiguous span of memory as an *artificial array*, using the binary operator '`@`'. The left operand of '`@`' should be the first element of the desired array and be an individual object. The right operand should be the desired length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program says

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

you can print the contents of `array` with

```
p *array@len
```

The left operand of '`@`' must reside in memory. Array values made with '`@`' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see Section 10.11 [Value History], page 161), after printing one out.

Another way to create an artificial array is to use a cast. This re-interprets a value as if it were an array. The value need not be in memory:

```
(gdb) p/x (short[2])0x12345678
```

```
$1 = {0x1234, 0x5678}
```

As a convenience, if you leave the array length out (as in '`(type[])value`') GDB calculates the size to fill the value (as '`sizeof(value)/sizeof(type)`':

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

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see Section 10.12 [Convenience Variables], page 162) as a counter in an expression that prints the first interesting value, and then repeat that expression via `RET`. For instance, suppose you have an array `dtab` of pointers to structures, and you are interested in the values of a field `fv` in each structure. Here is an example of what you might type:

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

10.5 Output Formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the `print` command with a slash and a format letter. The format letters supported are:

`x` Print the binary representation of the value in hexadecimal.

`d` Print the binary representation of the value in decimal.

`u` Print the binary representation of the value as an decimal, as if it were unsigned.

`o` Print the binary representation of the value in octal.

`t` Print the binary representation of the value in binary. The letter '`t`' stands for "two".[1]

`a` Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

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

The command `info symbol 0x54320` yields similar results. See Chapter 16 [Symbols], page 251.

[1] '`b`' cannot be used because these format letters are also used with the `x` command, where '`b`' stands for "byte"; see Section 10.6 [Examining Memory], page 143.

`c` Cast the value to an integer (unlike other formats, this does not just reinterpret the underlying bits) and print it as a character constant. This prints both the numerical value and its character representation. The character representation is replaced with the octal escape '`\nnn`' for characters outside the 7-bit ASCII range.

Without this format, GDB displays `char`, `unsigned char`, and `signed char` data as character constants. Single-byte members of vectors are displayed as integer data.

`f` Regard the bits of the value as a floating point number and print using typical floating point syntax.

`s` Regard as a string, if possible. With this format, pointers to single-byte data are displayed as null-terminated strings and arrays of single-byte data are displayed as fixed-length strings. Other values are displayed in their natural types.

Without this format, GDB displays pointers to and arrays of `char`, `unsigned char`, and `signed char` as strings. Single-byte members of a vector are displayed as an integer array.

`z` Like '`x`' formatting, the value is treated as an integer and printed as hexadecimal, but leading zeros are printed to pad the value to the size of the integer type.

`r` Print using the '`raw`' formatting. By default, GDB will use a Python-based pretty-printer, if one is available (see Section 10.10 [Pretty Printing], page 159). This typically results in a higher-level display of the value's contents. The '`r`' format bypasses any Python pretty-printer which might exist.

For example, to print the program counter in hex (see Section 10.14 [Registers], page 169), type

```
p/x $pc
```

Note that no space is required before the slash; this is because command names in GDB cannot contain a slash.

To reprint the last value in the value history with a different format, you can use the `print` command with just a format and no expression. For example, '`p/x`' reprints the last value in hex.

10.6 Examining Memory

You can use the command `x` (for "examine") to examine memory in any of several formats, independently of your program's data types.

`x/`*`nfu`* *`addr`*
`x` *`addr`*
`x` Use the `x` command to examine memory.

n, *f*, and *u* are all optional parameters that specify how much memory to display and how to format it; *addr* is an expression giving the address where you want to start displaying memory. If you use defaults for *nfu*, you need not type the slash '`/`'. Several commands set convenient defaults for *addr*.

n, the repeat count

The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units *u*) to display. If a negative number is specified, memory is examined backward from *addr*.

f, the display format

The display format is one of the formats used by `print` ('`x`', '`d`', '`u`', '`o`', '`t`', '`a`', '`c`', '`f`', '`s`'), '`i`' (for machine instructions) and '`m`' (for displaying memory tags). The default is '`x`' (hexadecimal) initially. The default changes each time you use either `x` or `print`.

u, the unit size

The unit size is any of

`b` Bytes.

`h` Halfwords (two bytes).

`w` Words (four bytes). This is the initial default.

`g` Giant words (eight bytes).

Each time you specify a unit size with `x`, that size becomes the default unit the next time you use `x`. For the '`i`' format, the unit size is ignored and is normally not written. For the '`s`' format, the unit size defaults to '`b`', unless it is explicitly given. Use `x /hs` to display 16-bit char strings and `x /ws` to display 32-bit strings. The next use of `x /s` will again display 8-bit strings. Note that the results depend on the programming language of the current compilation unit. If the language is C, the '`s`' modifier will use the UTF-16 encoding while '`w`' will use UTF-32. The encoding is set by the programming language and cannot be altered.

addr, starting display address

addr is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. See Section 10.1 [Expressions], page 137, for more information on expressions. The default for *addr* is usually just after the last address examined—but several other commands also set the default address: `info breakpoints` (to the address of the last breakpoint listed), `info line` (to the starting address of a line), and `print` (if you use it to display a value from memory).

For example, '`x/3uh 0x54320`' is a request to display three halfwords (`h`) of memory, formatted as unsigned decimal integers ('`u`'), starting at address `0x54320`. '`x/4xw $sp`' prints the four words ('`w`') of memory above the stack pointer (here, '`$sp`'; see Section 10.14 [Registers], page 169) in hexadecimal ('`x`').

You can also specify a negative repeat count to examine memory backward from the given address. For example, '`x/-3uh 0x54320`' prints three halfwords (`h`) at `0x5431a`, `0x5431c`, and `0x5431e`.

Since the letters indicating unit sizes are all distinct from the letters specifying output formats, you do not have to remember whether unit size or format comes first; either order

works. The output specifications '`4xw`' and '`4wx`' mean exactly the same thing. (However, the count *n* must come first; '`wx4`' does not work.)

Even though the unit size *u* is ignored for the formats '`s`' and '`i`', you might still want to use a count *n*; for example, '`3i`' specifies that you want to see three machine instructions, including any operands. For convenience, especially when used with the `display` command, the '`i`' format also prints branch delay slot instructions, if any, beyond the count specified, which immediately follow the last instruction that is within the count. The command `disassemble` gives an alternative way of inspecting machine instructions; see Section 9.6 [Source and Machine Code], page 128.

If a negative repeat count is specified for the formats '`s`' or '`i`', the command displays null-terminated strings or instructions before the given address as many as the absolute value of the given number. For the '`i`' format, we use line number information in the debug info to accurately locate instruction boundaries while disassembling backward. If line info is not available, the command stops examining memory with an error message.

All the defaults for the arguments to `x` are designed to make it easy to continue scanning memory with minimal specifications each time you use `x`. For example, after you have inspected three machine instructions with '`x/3i addr`', you can inspect the next seven with just '`x/7`'. If you use `RET` to repeat the `x` command, the repeat count *n* is used again; the other arguments default as for successive uses of `x`.

When examining machine instructions, the instruction at current program counter is shown with a `=>` marker. For example:

```
(gdb) x/5i $pc-6
   0x804837f <main+11>: mov    %esp,%ebp
   0x8048381 <main+13>: push   %ecx
   0x8048382 <main+14>: sub    $0x4,%esp
=> 0x8048385 <main+17>: movl   $0x8048460,(%esp)
   0x804838c <main+24>: call   0x80482d4 <puts@plt>
```

If the architecture supports memory tagging, the tags can be displayed by using '`m`'. See Section 10.7 [Memory Tagging], page 146.

The information will be displayed once per granule size (the amount of bytes a particular memory tag covers). For example, AArch64 has a granule size of 16 bytes, so it will display a tag every 16 bytes.

Due to the way GDB prints information with the `x` command (not aligned to a particular boundary), the tag information will refer to the initial address displayed on a particular line. If a memory tag boundary is crossed in the middle of a line displayed by the `x` command, it will be displayed on the next line.

The '`m`' format doesn't affect any other specified formats that were passed to the `x` command.

The addresses and contents printed by the `x` command are not saved in the value history because there is often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables `$_` and `$__`. After an `x` command, the last address examined is available for use in expressions in the convenience variable `$_`. The contents of that address, as examined, are available in the convenience variable `$__`.

If the `x` command has a repeat count, the address and contents saved are from the last memory unit printed; this is not the same as the last address printed if several units were printed on the last line of output.

Most targets have an addressable memory unit size of 8 bits. This means that to each memory address are associated 8 bits of data. Some targets, however, have other addressable memory unit sizes. Within GDB and this document, the term *addressable memory unit* (or *memory unit* for short) is used when explicitly referring to a chunk of data of that size. The word *byte* is used to refer to a chunk of data of 8 bits, regardless of the addressable memory unit size of the target. For most systems, addressable memory unit is a synonym of byte.

When you are debugging a program running on a remote target machine (see Chapter 20 [Remote Debugging], page 299), you may wish to verify the program's image in the remote machine's memory against the executable file you downloaded to the target. Or, on any target, you may want to check whether the program has corrupted its own read-only sections. The `compare-sections` command is provided for such situations.

`compare-sections` [*section-name*|`-r`]

Compare the data of a loadable section *section-name* in the executable file of the program being debugged with the same section in the target machine's memory, and report any mismatches. With no arguments, compares all loadable sections. With an argument of `-r`, compares all loadable read-only sections.

Note: for remote targets, this command can be accelerated if the target supports computing the CRC checksum of a block of memory (see [qCRC packet], page 700).

10.7 Memory Tagging

Memory tagging is a memory protection technology that uses a pair of tags to validate memory accesses through pointers. The tags are integer values usually comprised of a few bits, depending on the architecture.

There are two types of tags that are used in this setup: logical and allocation. A logical tag is stored in the pointers themselves, usually at the higher bits of the pointers. An allocation tag is the tag associated with particular ranges of memory in the physical address space, against which the logical tags from pointers are compared.

The pointer tag (logical tag) must match the memory tag (allocation tag) for the memory access to be valid. If the logical tag does not match the allocation tag, that will raise a memory violation.

Allocation tags cover multiple contiguous bytes of physical memory. This range of bytes is called a memory tag granule and is architecture-specific. For example, AArch64 has a tag granule of 16 bytes, meaning each allocation tag spans 16 bytes of memory.

If the underlying architecture supports memory tagging, like AArch64 MTE or SPARC ADI do, GDB can make use of it to validate pointers against memory allocation tags.

The `print` (see Chapter 10 [Data], page 133) and `x` (see Section 10.6 [Memory], page 143) commands will display tag information when appropriate, and a command prefix of `memory-tag` gives access to the various memory tagging commands.

The `memory-tag` commands are the following:

`memory-tag print-logical-tag` *`pointer_expression`*
: Print the logical tag stored in *pointer_expression*.

`memory-tag with-logical-tag` *`pointer_expression tag_bytes`*
: Print the pointer given by *pointer_expression*, augmented with a logical tag of *tag_bytes*.

`memory-tag print-allocation-tag` *`address_expression`*
: Print the allocation tag associated with the memory address given by *address_expression*.

`memory-tag setatag` *`starting_address length tag_bytes`*
: Set the allocation tag(s) for memory range [*starting_address*, *starting_address* + *length*) to *tag_bytes*.

`memory-tag check` *`pointer_expression`*
: Check if the logical tag in the pointer given by *pointer_expression* matches the allocation tag for the memory referenced by the pointer.

  This essentially emulates the hardware validation that is done when tagged memory is accessed through a pointer, but does not cause a memory fault as it would during hardware validation.

  It can be used to inspect potential memory tagging violations in the running process, before any faults get triggered.

10.8 Automatic Display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that GDB prints its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

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

This display shows item numbers, expressions and their current values. As with displays you request manually using `x` or `print`, you can specify the output format you prefer; in fact, `display` decides whether to use `print` or `x` depending your format specification—it uses `x` if you specify either the '`i`' or '`s`' format, or a unit size; otherwise it uses `print`.

`display` *`expr`*
: Add the expression *expr* to the list of expressions to display each time your program stops. See Section 10.1 [Expressions], page 137.

  `display` does not repeat if you press `RET` again after using it.

`display/`*`fmt expr`*
: For *fmt* specifying only a display format and not a size or count, add the expression *expr* to the auto-display list but arrange to display it each time in the specified format *fmt*. See Section 10.5 [Output Formats], page 142.

`display/`*`fmt addr`*
: For *fmt* '`i`' or '`s`', or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing '`x/`*`fmt addr`*'. See Section 10.6 [Examining Memory], page 143.

For example, '`display/i $pc`' can be helpful, to see the machine instruction about to be executed each time execution stops ('`$pc`' is a common name for the program counter; see Section 10.14 [Registers], page 169).

`undisplay` *`dnums...`*
`delete display` *`dnums...`*
: Remove items from the list of expressions to display. Specify the numbers of the displays that you want affected with the command argument *dnums*. It can be a single display number, one of the numbers shown in the first field of the '`info display`' display; or it could be a range of display numbers, as in `2-4`.

  `undisplay` does not repeat if you press `RET` after using it. (Otherwise you would just get the error '`No display number ...`'.)

`disable display` *`dnums...`*
: Disable the display of item numbers *dnums*. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later. Specify the numbers of the displays that you want affected with the command argument *dnums*. It can be a single display number, one of the numbers shown in the first field of the '`info display`' display; or it could be a range of display numbers, as in `2-4`.

`enable display` *`dnums...`*
: Enable display of item numbers *dnums*. It becomes effective once again in auto display of its expression, until you specify otherwise. Specify the numbers of the displays that you want affected with the command argument *dnums*. It can be a single display number, one of the numbers shown in the first field of the '`info display`' display; or it could be a range of display numbers, as in `2-4`.

`display`
: Display the current values of the expressions on the list, just as is done when your program stops.

`info display`
: Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which would not be displayed right now because they refer to automatic variables not currently available.

If a display expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is disabled when execution enters a context where one of its variables is not defined. For example, if you give the command `display last_char` while inside a function with an argument `last_char`, GDB displays this argument while your program continues to stop inside that function. When it stops elsewhere—where there is no variable `last_char`—the display is disabled automatically.

The next time your program stops where `last_char` is meaningful, you can enable the display expression once again.

10.9 Print Settings

GDB provides the following ways to control how arrays, structures, and symbols are printed.

These settings are useful for debugging programs in any language:

`set print address`
`set print address on`
: GDB prints memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is `on`. For example, this is what a stack frame display looks like with `set print address on`:

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

`set print address off`
: Do not print addresses when displaying their contents. For example, this is the same stack frame displayed with `set print address off`:

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

You can use '`set print address off`' to eliminate all machine dependent displays from the GDB interface. For example, with `print address off`, you should get the same text for backtraces on all machines—whether or not they involve pointer arguments.

`show print address`
: Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest earlier symbol plus an offset. If that symbol does not uniquely identify the address (for example, it is a name whose scope is a single source file), you may need to clarify. One way to do this is with `info line`, for example '`info line *0x4537`'. Alternately, you can set GDB to print the source file and line number when it prints a symbolic address:

`set print symbol-filename on`
: Tell GDB to print the source file name and line number of a symbol in the symbolic form of an address.

`set print symbol-filename off`
: Do not print source file name and line number of a symbol. This is the default.

`show print symbol-filename`
: Show whether or not GDB will print the source file name and line number of a symbol in the symbolic form of an address.

Another situation where it is helpful to show symbol filenames and line numbers is when disassembling code; GDB shows you the line number and source file that corresponds to each instruction.

Also, you may wish to see the symbolic form only if the address being printed is reasonably close to the closest earlier symbol:

`set print max-symbolic-offset` *max-offset*
`set print max-symbolic-offset unlimited`
: Tell GDB to only display the symbolic form of an address if the offset between the closest earlier symbol and the address is less than *max-offset*. The default is `unlimited`, which tells GDB to always print the symbolic form of an address if any symbol precedes it. Zero is equivalent to `unlimited`.

`show print max-symbolic-offset`
: Ask how large the maximum offset is that GDB prints in a symbolic address.

If you have a pointer and you are not sure where it points, try '`set print symbol-filename on`'. Then you can determine the name and source file location of the variable where it points, using '`p/a` *`pointer`*'. This interprets the address in symbolic form. For example, here GDB shows that a variable `ptt` points at another variable `t`, defined in `hi2.c`:

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

Warning: For pointers that point to a local variable, '`p/a`' does not show the symbol name and filename of the referent, even with the appropriate `set print` options turned on.

You can also enable '`/a`'-like formatting all the time using '`set print symbol on`':

`set print symbol on`
: Tell GDB to print the symbol corresponding to an address, if one exists.

`set print symbol off`
: Tell GDB not to print the symbol corresponding to an address. In this mode, GDB will still print the symbol corresponding to pointers to functions. This is the default.

`show print symbol`
: Show whether GDB will display the symbol corresponding to an address.

Other settings control how different kinds of objects are printed:

`set print array`
`set print array on`
: Pretty print arrays. This format is more convenient to read, but uses more space. The default is off.

`set print array off`
: Return to compressed format for arrays.

`show print array`
: Show whether compressed or pretty format is selected for displaying arrays.

`set print array-indexes`
`set print array-indexes on`
: Print the index of each element when displaying arrays. May be more convenient to locate a given element in the array or quickly find the index of a given element in that printed array. The default is off.

`set print array-indexes off`
: Stop printing element indexes when displaying arrays.

`show print array-indexes`
: Show whether the index of each element is printed when displaying arrays.

`set print elements` *`number-of-elements`*
`set print elements unlimited`
: Set a limit on how many elements of an array GDB will print. If GDB is printing a large array, it stops printing after it has printed the number of elements set by the `set print elements` command. This limit also applies to the display of strings. When GDB starts, this limit is set to 200. Setting *number-of-elements* to `unlimited` or zero means that the number of elements to print is unlimited.

`show print elements`
: Display the number of elements of a large array that GDB will print.

`set print frame-arguments` *`value`*
: This command allows to control how the values of arguments are printed when the debugger prints a frame (see Section 8.1 [Frames], page 107). The possible values are:

  `all` The values of all arguments are printed.

  `scalars` Print the value of an argument only if it is a scalar. The value of more complex arguments such as arrays, structures, unions, etc, is replaced by `...`. This is the default. Here is an example where only scalar arguments are shown:

```
#1  0x08048361 in call_me (i=3, s=..., ss=0xbf8d508c, u=..., e=green)
  at frame-args.c:23
```

  `none` None of the argument values are printed. Instead, the value of each argument is replaced by `...`. In this case, the example above now becomes:

```
#1  0x08048361 in call_me (i=..., s=..., ss=..., u=..., e=...)
  at frame-args.c:23
```

  `presence` Only the presence of arguments is indicated by `...`. The `...` are not printed for function without any arguments. None of the argument names and values are printed. In this case, the example above now becomes:

```
#1  0x08048361 in call_me (...) at frame-args.c:23
```

  By default, only scalar arguments are printed. This command can be used to configure the debugger to print the value of all arguments, regardless of their type. However, it is often advantageous to not print the value of more complex parameters. For instance, it reduces the amount of information printed in each

frame, making the backtrace more readable. Also, it improves performance when displaying Ada frames, because the computation of large arguments can sometimes be CPU-intensive, especially in large applications. Setting `print frame-arguments` to `scalars` (the default), `none` or `presence` avoids this computation, thus speeding up the display of each Ada frame.

`show print frame-arguments`
Show how the value of arguments should be displayed when printing a frame.

`set print raw-frame-arguments on`
Print frame arguments in raw, non pretty-printed, form.

`set print raw-frame-arguments off`
Print frame arguments in pretty-printed form, if there is a pretty-printer for the value (see Section 10.10 [Pretty Printing], page 159), otherwise print the value in raw form. This is the default.

`show print raw-frame-arguments`
Show whether to print frame arguments in raw form.

`set print entry-values` *value*
Set printing of frame argument values at function entry. In some cases GDB can determine the value of function argument which was passed by the function caller, even if the value was modified inside the called function and therefore is different. With optimized code, the current value could be unavailable, but the entry value may still be known.

The default value is `default` (see below for its description). Older GDB behaved as with the setting `no`. Compilers not supporting this feature will behave in the `default` setting the same way as with the `no` setting.

This functionality is currently supported only by DWARF 2 debugging format and the compiler has to produce '`DW_TAG_call_site`' tags. With GCC, you need to specify `-O -g` during compilation, to get this information.

The *value* parameter can be one of the following:

`no` Print only actual parameter values, never print values from function entry point.

```
#0  equal (val=5)
#0  different (val=6)
#0  lost (val=<optimized out>)
#0  born (val=10)
#0  invalid (val=<optimized out>)
```

`only` Print only parameter values from function entry point. The actual parameter values are never printed.

```
#0  equal (val@entry=5)
#0  different (val@entry=5)
#0  lost (val@entry=5)
#0  born (val@entry=<optimized out>)
#0  invalid (val@entry=<optimized out>)
```

`preferred`
: Print only parameter values from function entry point. If value from function entry point is not known while the actual value is known, print the actual value for such parameter.

```
#0  equal (val@entry=5)
#0  different (val@entry=5)
#0  lost (val@entry=5)
#0  born (val=10)
#0  invalid (val@entry=<optimized out>)
```

`if-needed`
: Print actual parameter values. If actual parameter value is not known while value from function entry point is known, print the entry point value for such parameter.

```
#0  equal (val=5)
#0  different (val=6)
#0  lost (val@entry=5)
#0  born (val=10)
#0  invalid (val=<optimized out>)
```

`both`
: Always print both the actual parameter value and its value from function entry point, even if values of one or both are not available due to compiler optimizations.

```
#0  equal (val=5, val@entry=5)
#0  different (val=6, val@entry=5)
#0  lost (val=<optimized out>, val@entry=5)
#0  born (val=10, val@entry=<optimized out>)
#0  invalid (val=<optimized out>, val@entry=<optimized out>)
```

`compact`
: Print the actual parameter value if it is known and also its value from function entry point if it is known. If neither is known, print for the actual value `<optimized out>`. If not in MI mode (see Chapter 27 [GDB/MI], page 537) and if both values are known and identical, print the shortened `param=param@entry=VALUE` notation.

```
#0  equal (val=val@entry=5)
#0  different (val=6, val@entry=5)
#0  lost (val@entry=5)
#0  born (val=10)
#0  invalid (val=<optimized out>)
```

`default`
: Always print the actual parameter value. Print also its value from function entry point, but only if it is known. If not in MI mode (see Chapter 27 [GDB/MI], page 537) and if both values are known and identical, print the shortened `param=param@entry=VALUE` notation.

```
#0  equal (val=val@entry=5)
#0  different (val=6, val@entry=5)
#0  lost (val=<optimized out>, val@entry=5)
#0  born (val=10)
#0  invalid (val=<optimized out>)
```

For analysis messages on possible failures of frame argument values at function entry resolution see [set debug entry-values], page 186.

`show print entry-values`
: Show the method being used for printing of frame argument values at function entry.

`set print frame-info` *value*
: This command allows to control the information printed when the debugger prints a frame. See Section 8.1 [Frames], page 107, Section 8.2 [Backtrace], page 108, for a general explanation about frames and frame information. Note that some other settings (such as `set print frame-arguments` and `set print address`) are also influencing if and how some frame information is displayed. In particular, the frame program counter is never printed if `set print address` is off.

  The possible values for `set print frame-info` are:

  `short-location`
  : Print the frame level, the program counter (if not at the beginning of the location source line), the function, the function arguments.

  `location`
  : Same as `short-location` but also print the source file and source line number.

  `location-and-address`
  : Same as `location` but print the program counter even if located at the beginning of the location source line.

  `source-line`
  : Print the program counter (if not at the beginning of the location source line), the line number and the source line.

  `source-and-location`
  : Print what `location` and `source-line` are printing.

  `auto`
  : The information printed for a frame is decided automatically by the GDB command that prints a frame. For example, `frame` prints the information printed by `source-and-location` while `stepi` will switch between `source-line` and `source-and-location` depending on the program counter. The default value is `auto`.

`set print repeats` *number-of-repeats*
`set print repeats unlimited`
: Set the threshold for suppressing display of repeated array elements. When the number of consecutive identical elements of an array exceeds the threshold, GDB prints the string `"<repeats` *n* `times>"`, where *n* is the number of identical repetitions, instead of displaying the identical elements themselves. Setting the threshold to `unlimited` or zero will cause all elements to be individually printed. The default threshold is 10.

`show print repeats`
: Display the current threshold for printing repeated identical elements.

`set print max-depth` *depth*
`set print max-depth unlimited`

Set the threshold after which nested structures are replaced with ellipsis, this can make visualising deeply nested structures easier.

For example, given this C code

```
typedef struct s1 { int a; } s1;
typedef struct s2 { s1 b; } s2;
typedef struct s3 { s2 c; } s3;
typedef struct s4 { s3 d; } s4;

s4 var = { { { { 3 } } } };
```

The following table shows how different values of *depth* will effect how `var` is printed by GDB:

depth **setting**	**Result of 'p var'**
unlimited	`$1 = {d = {c = {b = {a = 3}}}}`
0	`$1 = {...}`
1	`$1 = {d = {...}}`
2	`$1 = {d = {c = {...}}}`
3	`$1 = {d = {c = {b = {...}}}}`
4	`$1 = {d = {c = {b = {a = 3}}}}`

To see the contents of structures that have been hidden the user can either increase the print max-depth, or they can print the elements of the structure that are visible, for example

```
(gdb) set print max-depth 2
(gdb) p var
$1 = {d = {c = {...}}}
(gdb) p var.d
$2 = {c = {b = {...}}}
(gdb) p var.d.c
$3 = {b = {a = 3}}
```

The pattern used to replace nested structures varies based on language, for most languages `{...}` is used, but Fortran uses `(...)`.

`show print max-depth`

Display the current threshold after which nested structures are replaces with ellipsis.

`set print memory-tag-violations`
`set print memory-tag-violations on`

Cause GDB to display additional information about memory tag violations when printing pointers and addresses.

`set print memory-tag-violations off`

Stop printing memory tag violation information.

`show print memory-tag-violations`

Show whether memory tag violation information is displayed when printing pointers and addresses.

`set print null-stop`
: Cause GDB to stop printing the characters of an array when the first NULL is encountered. This is useful when large arrays actually contain only short strings. The default is off.

`show print null-stop`
: Show whether GDB stops printing an array on the first NULL character.

`set print pretty on`
: Cause GDB to print structures in an indented format with one member per line, like this:

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

`set print pretty off`
: Cause GDB to print structures in a compact format, like this:

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

: This is the default format.

`show print pretty`
: Show which format GDB is using to print structures.

`set print raw-values on`
: Print values in raw form, without applying the pretty printers for the value.

`set print raw-values off`
: Print values in pretty-printed form, if there is a pretty-printer for the value (see Section 10.10 [Pretty Printing], page 159), otherwise print the value in raw form.

: The default setting is "off".

`show print raw-values`
: Show whether to print values in raw form.

`set print sevenbit-strings on`
: Print using only seven-bit characters; if this option is set, GDB displays any eight-bit characters (in strings or character values) using the notation *nnn*. This setting is best if you are working in English (ASCII) and you use the high-order bit of characters as a marker or "meta" bit.

`set print sevenbit-strings off`
: Print full eight-bit characters. This allows the use of more international character sets, and is the default.

`show print sevenbit-strings`
: Show whether or not GDB is printing only seven-bit characters.

`set print union on`
: Tell GDB to print unions which are contained in structures and other unions. This is the default setting.

`set print union off`
: Tell GDB not to print unions which are contained in structures and other unions. GDB will print `"{...}"` instead.

`show print union`
: Ask GDB whether or not it will print unions which are contained in structures and other unions.

  For example, given the declarations

```
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}};
```

  with `set print union on` in effect '`p foo`' would print

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

  and with `set print union off` in effect it would print

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

  `set print union` affects programs written in C-like languages and in Pascal.

These settings are of interest when debugging C++ programs:

`set print demangle`
`set print demangle on`
: Print C++ names in their source form rather than in the encoded ("mangled") form passed to the assembler and linker for type-safe linkage. The default is on.

`show print demangle`
: Show whether C++ names are printed in mangled or demangled form.

`set print asm-demangle`
`set print asm-demangle on`
: Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

`show print asm-demangle`
: Show whether C++ names in assembly listings are printed in mangled or demangled form.

`set demangle-style` *style*
: Choose among several encoding schemes used by different compilers to represent C++ names. If you omit *style*, you will see a list of possible formats. The default value is *auto*, which lets GDB choose a decoding style by inspecting your program.

`show demangle-style`
: Display the encoding style currently in use for decoding C++ symbols.

`set print object`
`set print object on`
: When displaying a pointer to an object, identify the *actual* (derived) type of the object rather than the *declared* type, using the virtual function table. Note that the virtual function table is required—this feature can only work for objects that have run-time type identification; a single virtual method in the object's declared type is sufficient. Note that this setting is also taken into account when working with variable objects via MI (see Chapter 27 [GDB/MI], page 537).

`set print object off`
: Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

`show print object`
: Show whether actual, or declared, object types are displayed.

`set print static-members`
`set print static-members on`
: Print static members when displaying a C++ object. The default is on.

`set print static-members off`
: Do not print static members when displaying a C++ object.

`show print static-members`
: Show whether C++ static members are printed or not.

`set print pascal_static-members`
`set print pascal_static-members on`
: Print static members when displaying a Pascal object. The default is on.

`set print pascal_static-members off`
: Do not print static members when displaying a Pascal object.

`show print pascal_static-members`
: Show whether Pascal static members are printed or not.

`set print vtbl`
`set print vtbl on`
: Pretty print C++ virtual function tables. The default is off. (The `vtbl` commands do not work on programs compiled with the HP ANSI C++ compiler (`aCC`).)

`set print vtbl off`
: Do not pretty print C++ virtual function tables.

`show print vtbl`
: Show whether C++ virtual function tables are pretty printed, or not.

10.10 Pretty Printing

GDB provides a mechanism to allow pretty-printing of values using Python code. It greatly simplifies the display of complex objects. This mechanism works for both MI and the CLI.

10.10.1 Pretty-Printer Introduction

When GDB prints a value, it first sees if there is a pretty-printer registered for the value. If there is then GDB invokes the pretty-printer to print the value. Otherwise the value is printed normally.

Pretty-printers are normally named. This makes them easy to manage. The '`info pretty-printer`' command will list all the installed pretty-printers with their names. If a pretty-printer can handle multiple data types, then its *subprinters* are the printers for the individual data types. Each such subprinter has its own name. The format of the name is *printer-name*;*subprinter-name*.

Pretty-printers are installed by *registering* them with GDB. Typically they are automatically loaded and registered when the corresponding debug information is loaded, thus making them available without having to do anything special.

There are three places where a pretty-printer can be registered.

- Pretty-printers registered globally are available when debugging all inferiors.
- Pretty-printers registered with a program space are available only when debugging that program. See Section 23.3.2.24 [Progspaces In Python], page 439, for more details on program spaces in Python.
- Pretty-printers registered with an objfile are loaded and unloaded with the corresponding objfile (e.g., shared library). See Section 23.3.2.25 [Objfiles In Python], page 441, for more details on objfiles in Python.

See Section 23.3.2.6 [Selecting Pretty-Printers], page 398, for further information on how pretty-printers are selected,

See Section 23.3.2.7 [Writing a Pretty-Printer], page 399, for implementing pretty printers for new types.

10.10.2 Pretty-Printer Example

Here is how a C++ `std::string` looks without a pretty-printer:

```
(gdb) print s
$1 = {
  static npos = 4294967295,
  _M_dataplus = {
    <std::allocator<char>> = {
      <__gnu_cxx::new_allocator<char>> = {
        <No data fields>}, <No data fields>
      },
    members of std::basic_string<char, std::char_traits<char>,
      std::allocator<char> >::_Alloc_hider:
    _M_p = 0x804a014 "abcd"
  }
```

```
}
```

With a pretty-printer for `std::string` only the contents are printed:

```
(gdb) print s
$2 = "abcd"
```

10.10.3 Pretty-Printer Commands

`info pretty-printer [`*object-regexp* `[`*name-regexp*`]]`

Print the list of installed pretty-printers. This includes disabled pretty-printers, which are marked as such.

object-regexp is a regular expression matching the objects whose pretty-printers to list. Objects can be `global`, the program space's file (see Section 23.3.2.24 [Progspaces In Python], page 439), and the object files within that program space (see Section 23.3.2.25 [Objfiles In Python], page 441). See Section 23.3.2.6 [Selecting Pretty-Printers], page 398, for details on how GDB looks up a printer from these three objects.

name-regexp is a regular expression matching the name of the printers to list.

`disable pretty-printer [`*object-regexp* `[`*name-regexp*`]]`

Disable pretty-printers matching *object-regexp* and *name-regexp*. A disabled pretty-printer is not forgotten, it may be enabled again later.

`enable pretty-printer [`*object-regexp* `[`*name-regexp*`]]`

Enable pretty-printers matching *object-regexp* and *name-regexp*.

Example:

Suppose we have three pretty-printers installed: one from library1.so named `foo` that prints objects of type `foo`, and another from library2.so named `bar` that prints two types of objects, `bar1` and `bar2`.

```
(gdb) info pretty-printer
library1.so:
  foo
library2.so:
  bar
    bar1
    bar2
(gdb) info pretty-printer library2
library2.so:
  bar
    bar1
    bar2
(gdb) disable pretty-printer library1
1 printer disabled
2 of 3 printers enabled
(gdb) info pretty-printer
library1.so:
  foo [disabled]
library2.so:
  bar
    bar1
    bar2
(gdb) disable pretty-printer library2 bar;bar1
1 printer disabled
```

```
1 of 3 printers enabled
(gdb) info pretty-printer library2
library1.so:
  foo [disabled]
library2.so:
  bar
    bar1 [disabled]
    bar2
(gdb) disable pretty-printer library2 bar
1 printer disabled
0 of 3 printers enabled
(gdb) info pretty-printer library2
library1.so:
  foo [disabled]
library2.so:
  bar [disabled]
    bar1 [disabled]
    bar2
```

Note that for `bar` the entire printer can be disabled, as can each individual subprinter.

Printing values and frame arguments is done by default using the enabled pretty printers.

The print option `-raw-values` and GDB setting `set print raw-values` (see [set print raw-values], page 156) can be used to print values without applying the enabled pretty printers.

Similarly, the backtrace option `-raw-frame-arguments` and GDB setting `set print raw-frame-arguments` (see [set print raw-frame-arguments], page 152) can be used to ignore the enabled pretty printers when printing frame argument values.

10.11 Value History

Values printed by the `print` command are saved in the GDB *value history*. This allows you to refer to them in other expressions. Values are kept until the symbol table is re-read or discarded (for example with the `file` or `symbol-file` commands). When the symbol table changes, the value history is discarded, since the values may contain pointers back to the types defined in the symbol table.

The values printed are given *history numbers* by which you can refer to them. These are successive integers starting with one. `print` shows you the history number assigned to a value by printing '`$`*num* `= `' before the value; here *num* is the history number.

To refer to any previous value, use '`$`' followed by the value's history number. The way `print` labels its output is designed to remind you of this. Just `$` refers to the most recent value in the history, and `$$` refers to the value before that. `$$`*n* refers to the *n*th value from the end; `$$2` is the value just prior to `$$`, `$$1` is equivalent to `$$`, and `$$0` is equivalent to `$`.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type

```
p *$
```

If you have a chain of structures where the component `next` points to the next one, you can print the contents of the next one with this:

```
p *$.next
```

You can print successive links in the chain by repeating this command—which you can do by just typing `RET`.

Note that the history records values, not expressions. If the value of `x` is 4 and you type these commands:

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

then the value recorded in the value history by the `print` command remains 4 even though the value of `x` has changed.

`show values`
: Print the last ten values in the value history, with their item numbers. This is like '`p $$9`' repeated ten times, except that `show values` does not change the history.

`show values` *n*
: Print ten history values centered on history item number *n*.

`show values +`
: Print ten history values just after the values last printed. If no more values are available, `show values +` produces no display.

Pressing `RET` to repeat `show values` *n* has exactly the same effect as '`show values +`'.

10.12 Convenience Variables

GDB provides *convenience variables* that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no direct effect on further execution of your program. That is why you can use them freely.

Convenience variables are prefixed with '`$`'. Any name preceded by '`$`' can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see Section 10.14 [Registers], page 169). (Value history references, in contrast, are *numbers* preceded by '`$`'. See Section 10.11 [Value History], page 161.)

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. For example:

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

would save in `$foo` the value contained in the object pointed to by `object_ptr`.

Using a convenience variable for the first time creates it, but its value is `void` until you assign a new value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

`show convenience`
: Print a list of convenience variables used so far, and their values, as well as a list of the convenience functions. Abbreviated `show conv`.

`init-if-undefined` `$`*variable* `=` *expression*

Set a convenience variable if it has not already been set. This is useful for user-defined commands that keep some state. It is similar, in concept, to using local static variables with initializers in C (except that convenience variables are global). It can also be used to allow users to override default values used in a command script.

If the variable is already defined then the expression is not evaluated so any side-effects do not occur.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:

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

Repeat that command by typing `RET`.

Some convenience variables are created automatically by GDB and given values likely to be useful.

`$_` The variable `$_` is automatically set by the `x` command to the last address examined (see Section 10.6 [Examining Memory], page 143). Other commands which provide a default address for `x` to examine also set `$_` to that address; these commands include `info line` and `info breakpoint`. The type of `$_` is `void *` except when set by the `x` command, in which case it is a pointer to the type of `$__`.

`$__` The variable `$__` is automatically set by the `x` command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.

`$_exitcode`

When the program being debugged terminates normally, GDB automatically sets this variable to the exit code of the program, and resets `$_exitsignal` to `void`.

`$_exitsignal`

When the program being debugged dies due to an uncaught signal, GDB automatically sets this variable to that signal's number, and resets `$_exitcode` to `void`.

To distinguish between whether the program being debugged has exited (i.e., `$_exitcode` is not `void`) or signalled (i.e., `$_exitsignal` is not `void`), the convenience function `$_isvoid` can be used (see Section 10.13 [Convenience Functions], page 165). For example, considering the following source code:

```
#include <signal.h>

int
main (int argc, char *argv[])
{
  raise (SIGALRM);
  return 0;
}
```

A valid way of telling whether the program being debugged has exited or signalled would be:

```
(gdb) define has_exited_or_signalled
Type commands for definition of ‘‘has_exited_or_signalled’’.
End with a line saying just ‘‘end’’.
>if $_isvoid ($_exitsignal)
 >echo The program has exited\n
 >else
 >echo The program has signalled\n
 >end
>end
(gdb) run
Starting program:

Program terminated with signal SIGALRM, Alarm clock.
The program no longer exists.
(gdb) has_exited_or_signalled
The program has signalled
```

As can be seen, GDB correctly informs that the program being debugged has signalled, since it calls `raise` and raises a `SIGALRM` signal. If the program being debugged had not called `raise`, then GDB would report a normal exit:

```
(gdb) has_exited_or_signalled
The program has exited
```

`$_exception`
: The variable `$_exception` is set to the exception object being thrown at an exception-related catchpoint. See Section 5.1.3 [Set Catchpoints], page 66.

`$_ada_exception`
: The variable `$_ada_exception` is set to the address of the exception being caught or thrown at an Ada exception-related catchpoint. See Section 5.1.3 [Set Catchpoints], page 66.

`$_probe_argc`
`$_probe_arg0...$_probe_arg11`
: Arguments to a static probe. See Section 5.1.10 [Static Probe Points], page 77.

`$_sdata` The variable `$_sdata` contains extra collected static tracepoint data. See Section 13.1.6 [Tracepoint Action Lists], page 198. Note that `$_sdata` could be empty, if not inspecting a trace buffer, or if extra static tracepoint data has not been collected.

`$_siginfo`
: The variable `$_siginfo` contains extra signal information (see [extra signal information], page 88). Note that `$_siginfo` could be empty, if the application has not yet received any signals. For example, it will be empty before you execute the `run` command.

`$_tlb` The variable `$_tlb` is automatically set when debugging applications running on MS-Windows in native mode or connected to gdbserver that supports the `qGetTIBAddr` request. See Section E.4 [General Query Packets], page 699. This variable contains the address of the thread information block.

`$_inferior`
: The number of the current inferior. See Section 4.9 [Debugging Multiple Inferiors Connections and Programs], page 40.

`$_thread` The thread number of the current thread. See [thread numbers], page 45.

`$_gthread`
: The global number of the current thread. See [global thread numbers], page 45.

`$_gdb_major`
`$_gdb_minor`
: The major and minor version numbers of the running GDB. Development snapshots and pretest versions have their minor version incremented by one; thus, GDB pretest 9.11.90 will produce the value 12 for `$_gdb_minor`. These variables allow you to write scripts that work with different versions of GDB without errors caused by features unavailable in some of those versions.

`$_shell_exitcode`
`$_shell_exitsignal`
: GDB commands such as `shell` and `|` are launching shell commands. When a launched command terminates, GDB automatically maintains the variables `$_shell_exitcode` and `$_shell_exitsignal` according to the exit status of the last launched command. These variables are set and used similarly to the variables `$_exitcode` and `$_exitsignal`.

10.13 Convenience Functions

GDB also supplies some *convenience functions*. These have a syntax similar to convenience variables. A convenience function can be used in an expression just like an ordinary function; however, a convenience function is implemented internally to GDB.

These functions do not require GDB to be configured with `Python` support, which means that they are always available.

`$_isvoid (`*expr*`)`
: Return one if the expression *expr* is `void`. Otherwise it returns zero.

  A `void` expression is an expression where the type of the result is `void`. For example, you can examine a convenience variable (see Section 10.12 [Convenience Variables], page 162) to check whether it is `void`:

```
(gdb) print $_exitcode
$1 = void
(gdb) print $_isvoid ($_exitcode)
$2 = 1
(gdb) run
Starting program: ./a.out
[Inferior 1 (process 29572) exited normally]
(gdb) print $_exitcode
$3 = 0
(gdb) print $_isvoid ($_exitcode)
$4 = 0
```

  In the example above, we used `$_isvoid` to check whether `$_exitcode` is `void` before and after the execution of the program being debugged. Before the

execution there is no exit code to be examined, therefore `$_exitcode` is `void`. After the execution the program being debugged returned zero, therefore `$_exitcode` is zero, which means that it is not `void` anymore.

The `void` expression can also be a call of a function from the program being debugged. For example, given the following function:

```
void
foo (void)
{
}
```

The result of calling it inside GDB is `void`:

```
(gdb) print foo ()
$1 = void
(gdb) print $_isvoid (foo ())
$2 = 1
(gdb) set $v = foo ()
(gdb) print $v
$3 = void
(gdb) print $_isvoid ($v)
$4 = 1
```

`$_gdb_setting_str (`*setting*`)`

Return the value of the GDB *setting* as a string. *setting* is any setting that can be used in a `set` or `show` command (see Chapter 22 [Controlling GDB], page 345).

```
(gdb) show print frame-arguments
Printing of non-scalar frame arguments is "scalars".
(gdb) p $_gdb_setting_str("print frame-arguments")
$1 = "scalars"
(gdb) p $_gdb_setting_str("height")
$2 = "30"
(gdb)
```

`$_gdb_setting (`*setting*`)`

Return the value of the GDB *setting*. The type of the returned value depends on the setting.

The value type for boolean and auto boolean settings is `int`. The boolean values `off` and `on` are converted to the integer values `0` and `1`. The value `auto` is converted to the value `-1`.

The value type for integer settings is either `unsigned int` or `int`, depending on the setting.

Some integer settings accept an `unlimited` value. Depending on the setting, the `set` command also accepts the value `0` or the value -1 as a synonym for `unlimited`. For example, `set height unlimited` is equivalent to `set height 0`.

Some other settings that accept the `unlimited` value use the value `0` to literally mean zero. For example, `set history size 0` indicates to not record any GDB commands in the command history. For such settings, -1 is the synonym for `unlimited`.

See the documentation of the corresponding `set` command for the numerical value equivalent to `unlimited`.

The `$_gdb_setting` function converts the unlimited value to a `0` or a `−1` value according to what the `set` command uses.

```
(gdb) p $_gdb_setting_str("height")
$1 = "30"
(gdb) p $_gdb_setting("height")
$2 = 30
(gdb) set height unlimited
(gdb) p $_gdb_setting_str("height")
$3 = "unlimited"
(gdb) p $_gdb_setting("height")
$4 = 0
(gdb) p $_gdb_setting_str("history size")
$5 = "unlimited"
(gdb) p $_gdb_setting("history size")
$6 = -1
(gdb) p $_gdb_setting_str("disassemble-next-line")
$7 = "auto"
(gdb) p $_gdb_setting("disassemble-next-line")
$8 = -1
(gdb)
```

Other setting types (enum, filename, optional filename, string, string noescape) are returned as string values.

`$_gdb_maint_setting_str (`*`setting`*`)`
: Like the `$_gdb_setting_str` function, but works with `maintenance set` variables.

`$_gdb_maint_setting (`*`setting`*`)`
: Like the `$_gdb_setting` function, but works with `maintenance set` variables.

The following functions require GDB to be configured with `Python` support.

`$_memeq(`*`buf1`*`, `*`buf2`*`, `*`length`*`)`
: Returns one if the *length* bytes at the addresses given by *buf1* and *buf2* are equal. Otherwise it returns zero.

`$_regex(`*`str`*`, `*`regex`*`)`
: Returns one if the string *str* matches the regular expression *regex*. Otherwise it returns zero. The syntax of the regular expression is that specified by `Python`'s regular expression support.

`$_streq(`*`str1`*`, `*`str2`*`)`
: Returns one if the strings *str1* and *str2* are equal. Otherwise it returns zero.

`$_strlen(`*`str`*`)`
: Returns the length of string *str*.

`$_caller_is(`*`name`*`[, `*`number_of_frames`*`])`
: Returns one if the calling function's name is equal to *name*. Otherwise it returns zero.

  If the optional argument *number_of_frames* is provided, it is the number of frames up in the stack to look. The default is 1.

  Example:

```
(gdb) backtrace
```

```
#0  bottom_func ()
    at testsuite/gdb.python/py-caller-is.c:21
#1  0x00000000004005a0 in middle_func ()
    at testsuite/gdb.python/py-caller-is.c:27
#2  0x00000000004005ab in top_func ()
    at testsuite/gdb.python/py-caller-is.c:33
#3  0x00000000004005b6 in main ()
    at testsuite/gdb.python/py-caller-is.c:39
(gdb) print $_caller_is ("middle_func")
$1 = 1
(gdb) print $_caller_is ("top_func", 2)
$1 = 1
```

`$_caller_matches(`*regexp*`[, `*number_of_frames*`])`

Returns one if the calling function's name matches the regular expression *regexp*. Otherwise it returns zero.

If the optional argument *number_of_frames* is provided, it is the number of frames up in the stack to look. The default is 1.

`$_any_caller_is(`*name*`[, `*number_of_frames*`])`

Returns one if any calling function's name is equal to *name*. Otherwise it returns zero.

If the optional argument *number_of_frames* is provided, it is the number of frames up in the stack to look. The default is 1.

This function differs from `$_caller_is` in that this function checks all stack frames from the immediate caller to the frame specified by *number_of_frames*, whereas `$_caller_is` only checks the frame specified by *number_of_frames*.

`$_any_caller_matches(`*regexp*`[, `*number_of_frames*`])`

Returns one if any calling function's name matches the regular expression *regexp*. Otherwise it returns zero.

If the optional argument *number_of_frames* is provided, it is the number of frames up in the stack to look. The default is 1.

This function differs from `$_caller_matches` in that this function checks all stack frames from the immediate caller to the frame specified by *number_of_frames*, whereas `$_caller_matches` only checks the frame specified by *number_of_frames*.

`$_as_string(`*value*`)`

Return the string representation of *value*.

This function is useful to obtain the textual label (enumerator) of an enumeration value. For example, assuming the variable *node* is of an enumerated type:

```
(gdb) printf "Visiting node of type %s\n", $_as_string(node)
Visiting node of type NODE_INTEGER
```

`$_cimag(`*value*`)`
`$_creal(`*value*`)`

Return the imaginary (`$_cimag`) or real (`$_creal`) part of the complex number *value*.

The type of the imaginary or real part depends on the type of the complex number, e.g., using `$_cimag` on a `float complex` will return an imaginary part of type `float`.

GDB provides the ability to list and get help on convenience functions.

`help function`
: Print a list of all convenience functions.

10.14 Registers

You can refer to machine register contents, in expressions, as variables with names starting with '`$`'. The names of registers are different for each machine; use `info registers` to see the names used on your machine.

`info registers`
: Print the names and values of all registers except floating-point and vector registers (in the selected stack frame).

`info all-registers`
: Print the names and values of all registers, including floating-point and vector registers (in the selected stack frame).

`info registers` *reggroup* `...`
: Print the name and value of the registers in each of the specified *reggroup*s. The *reggroup* can be any of those returned by `maint print reggroups` (see Appendix D [Maintenance Commands], page 671).

`info registers` *regname* `...`
: Print the *relativized* value of each specified register *regname*. As discussed in detail below, register values are normally relative to the selected stack frame. The *regname* may be any register name valid on the machine you are using, with or without the initial '`$`'.

GDB has four "standard" register names that are available (in expressions) on most machines—whenever they do not conflict with an architecture's canonical mnemonics for registers. The register names `$pc` and `$sp` are used for the program counter register and the stack pointer. `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. For example, you could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer[2] with

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

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict.

[2] This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use `return`; see Section 17.4 [Returning from a Function], page 268.

The `info registers` command shows the canonical names. For example, on the SPARC, `info registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`; and on x86-based machines `$ps` is an alias for the EFLAGS register.

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered to have floating point values. There is no way to refer to the contents of an ordinary register as floating point value (although you can *print* it as a floating point value with '`print/f $regname`').

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in "extended" (raw) format, but all C programs expect to work with "double" (virtual) format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the `info registers` command prints the data in both formats.

Some machines have special registers whose contents can be interpreted in several different ways. For example, modern x86-based machines have SSE and MMX registers that can hold several values packed together in several different formats. GDB refers to such registers in `struct` notation:

```
(gdb) print $xmm1
$1 = {
  v4_float = {0, 3.43859137e-038, 1.54142831e-044, 1.821688e-044},
  v2_double = {9.92129282474342e-303, 2.7585945287983262e-313},
  v16_int8 = "\000\000\000\000\3706;\001\v\000\000\000\r\000\000",
  v8_int16 = {0, 0, 14072, 315, 11, 0, 13, 0},
  v4_int32 = {0, 20657912, 11, 13},
  v2_int64 = {88725056443645952, 55834574859},
  uint128 = 0x0000000d0000000b013b36f800000000
}
```

To set values of such registers, you need to tell GDB which view of the register you wish to change, as if you were assigning value to a `struct` member:

```
(gdb) set $xmm1.uint128 = 0x00000000000000000000000FFFFFFFF
```

Normally, register values are relative to the selected stack frame (see Section 8.3 [Selecting a Frame], page 111). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with '`frame 0`').

Usually ABIs reserve some registers as not needed to be saved by the callee (a.k.a.: "caller-saved", "call-clobbered" or "volatile" registers). It may therefore not be possible for GDB to know the value a register had before the call (in other words, in the outer frame), if the register value has since been changed by the callee. GDB tries to deduce where the inner frame saved ("callee-saved") registers, from the debug info, unwind info, or the machine code generated by your compiler. If some register is not saved, and GDB knows the register is "caller-saved" (via its own knowledge of the ABI, or because the debug/unwind info explicitly says the register's value is undefined), GDB displays '`<not saved>`' as the register's value. With targets that GDB has no knowledge of the register saving convention, if a register was not saved by the callee, then its value and location in the outer frame are

assumed to be the same of the inner frame. This is usually harmless, because if the register is call-clobbered, the caller either does not care what is in the register after the call, or has code to restore the value that it does care about. Note, however, that if you change such a register in the outer frame, you may also be affecting the inner frame. Also, the more "outer" the frame is you're looking at, the more likely a call-clobbered register's value is to be wrong, in the sense that it doesn't actually represent the value the register had just before the call.

10.15 Floating Point Hardware

Depending on the configuration, GDB may be able to give you more information about the status of the floating point hardware.

`info float`
: Display hardware-dependent information about the floating point unit. The exact contents and layout vary depending on the floating point chip. Currently, '`info float`' is supported on the ARM and x86 machines.

10.16 Vector Unit

Depending on the configuration, GDB may be able to give you more information about the status of the vector unit.

`info vector`
: Display information about the vector unit. The exact contents and layout vary depending on the hardware.

10.17 Operating System Auxiliary Information

GDB provides interfaces to useful OS facilities that can help you debug your program.

Some operating systems supply an *auxiliary vector* to programs at startup. This is akin to the arguments and environment that you specify for a program, but contains a system-dependent variety of binary values that tell system libraries important details about the hardware, operating system, and process. Each value's purpose is identified by an integer tag; the meanings are well-known but system-specific. Depending on the configuration and operating system facilities, GDB may be able to show you this information. For remote targets, this functionality may further depend on the remote stub's support of the '`qXfer:auxv:read`' packet, see [qXfer auxiliary vector read], page 719.

`info auxv` Display the auxiliary vector of the inferior, which can be either a live process or a core dump file. GDB prints each tag value numerically, and also shows names and text descriptions for recognized tags. Some values in the vector are numbers, some bit masks, and some pointers to strings or other data. GDB displays each value in the most appropriate form for a recognized tag, and in hexadecimal for an unrecognized tag.

On some targets, GDB can access operating system-specific information and show it to you. The types of information available will differ depending on the type of operating system running on the target. The mechanism used to fetch the data is described in Appendix H [Operating System Information], page 783. For remote targets, this functionality depends

on the remote stub's support of the '`qXfer:osdata:read`' packet, see [qXfer osdata read], page 722.

`info os` *infotype*

Display OS information of the requested type.

On GNU/Linux, the following values of *infotype* are valid:

`cpus` Display the list of all CPUs/cores. For each CPU/core, GDB prints the available fields from /proc/cpuinfo. For each supported architecture different fields are available. Two common entries are processor which gives CPU number and bogomips; a system constant that is calculated during kernel initialization.

`files` Display the list of open file descriptors on the target. For each file descriptor, GDB prints the identifier of the process owning the descriptor, the command of the owning process, the value of the descriptor, and the target of the descriptor.

`modules` Display the list of all loaded kernel modules on the target. For each module, GDB prints the module name, the size of the module in bytes, the number of times the module is used, the dependencies of the module, the status of the module, and the address of the loaded module in memory.

`msg` Display the list of all System V message queues on the target. For each message queue, GDB prints the message queue key, the message queue identifier, the access permissions, the current number of bytes on the queue, the current number of messages on the queue, the processes that last sent and received a message on the queue, the user and group of the owner and creator of the message queue, the times at which a message was last sent and received on the queue, and the time at which the message queue was last changed.

`processes`
Display the list of processes on the target. For each process, GDB prints the process identifier, the name of the user, the command corresponding to the process, and the list of processor cores that the process is currently running on. (To understand what these properties mean, for this and the following info types, please consult the general GNU/Linux documentation.)

`procgroups`
Display the list of process groups on the target. For each process, GDB prints the identifier of the process group that it belongs to, the command corresponding to the process group leader, the process identifier, and the command line of the process. The list is sorted first by the process group identifier, then by the process identifier, so that processes belonging to the same process group are grouped together and the process group leader is listed first.

`semaphores`
: Display the list of all System V semaphore sets on the target. For each semaphore set, GDB prints the semaphore set key, the semaphore set identifier, the access permissions, the number of semaphores in the set, the user and group of the owner and creator of the semaphore set, and the times at which the semaphore set was operated upon and changed.

`shm`
: Display the list of all System V shared-memory regions on the target. For each shared-memory region, GDB prints the region key, the shared-memory identifier, the access permissions, the size of the region, the process that created the region, the process that last attached to or detached from the region, the current number of live attaches to the region, and the times at which the region was last attached to, detach from, and changed.

`sockets`
: Display the list of Internet-domain sockets on the target. For each socket, GDB prints the address and port of the local and remote endpoints, the current state of the connection, the creator of the socket, the IP address family of the socket, and the type of the connection.

`threads`
: Display the list of threads running on the target. For each thread, GDB prints the identifier of the process that the thread belongs to, the command of the process, the thread identifier, and the processor core that it is currently running on. The main thread of a process is not listed.

`info os`
: If *infotype* is omitted, then list the possible values for *infotype* and the kind of OS information available for each *infotype*. If the target does not return a list of possible types, this command will report an error.

10.18 Memory Region Attributes

Memory region attributes allow you to describe special handling required by regions of your target's memory. GDB uses attributes to determine whether to allow certain types of memory accesses; whether to use specific width accesses; and whether to cache target memory. By default the description of memory regions is fetched from the target (if the current target supports this), but the user can override the fetched regions.

Defined memory regions can be individually enabled and disabled. When a memory region is disabled, GDB uses the default attributes when accessing memory in that region. Similarly, if no memory regions have been defined, GDB uses the default attributes when accessing all memory.

When a memory region is defined, it is given a number to identify it; to enable, disable, or remove a memory region, you specify that number.

`mem` *`lower upper attributes...`*
: Define a memory region bounded by *lower* and *upper* with attributes *attributes*..., and add it to the list of regions monitored by GDB. Note that

upper == 0 is a special case: it is treated as the target's maximum memory address. (0xffff on 16 bit targets, 0xffffffff on 32 bit targets, etc.)

`mem auto` Discard any user changes to the memory regions and use target-supplied regions, if available, or no regions if the target does not support.

`delete mem` *nums...*
Remove memory regions *nums*... from the list of regions monitored by GDB.

`disable mem` *nums...*
Disable monitoring of memory regions *nums*.... A disabled memory region is not forgotten. It may be enabled again later.

`enable mem` *nums...*
Enable monitoring of memory regions *nums*....

`info mem` Print a table of all defined memory regions, with the following columns for each region:

Memory Region Number
Enabled or Disabled.
Enabled memory regions are marked with '`y`'. Disabled memory regions are marked with '`n`'.

Lo Address
The address defining the inclusive lower bound of the memory region.

Hi Address
The address defining the exclusive upper bound of the memory region.

Attributes The list of attributes set for this memory region.

10.18.1 Attributes

10.18.1.1 Memory Access Mode

The access mode attributes set whether GDB may make read or write accesses to a memory region.

While these attributes prevent GDB from performing invalid memory accesses, they do nothing to prevent the target system, I/O DMA, etc. from accessing memory.

`ro` Memory is read only.

`wo` Memory is write only.

`rw` Memory is read/write. This is the default.

10.18.1.2 Memory Access Size

The access size attribute tells GDB to use specific sized accesses in the memory region. Often memory mapped device registers require specific sized accesses. If no access size attribute is specified, GDB may use accesses of any size.

`8` Use 8 bit memory accesses.

`16` Use 16 bit memory accesses.

`32` Use 32 bit memory accesses.

`64` Use 64 bit memory accesses.

10.18.1.3 Data Cache

The data cache attributes set whether GDB will cache target memory. While this generally improves performance by reducing debug protocol overhead, it can lead to incorrect results because GDB does not know about volatile variables or memory mapped device registers.

`cache` Enable GDB to cache target memory.

`nocache` Disable GDB from caching target memory. This is the default.

10.18.2 Memory Access Checking

GDB can be instructed to refuse accesses to memory that is not explicitly described. This can be useful if accessing such regions has undesired effects for a specific target, or to provide better error checking. The following commands control this behaviour.

`set mem inaccessible-by-default [on|off]`
: If `on` is specified, make GDB treat memory not explicitly described by the memory ranges as non-existent and refuse accesses to such memory. The checks are only performed if there's at least one memory range defined. If `off` is specified, make GDB treat the memory not explicitly described by the memory ranges as RAM. The default value is `on`.

`show mem inaccessible-by-default`
: Show the current handling of accesses to unknown memory.

10.19 Copy Between Memory and a File

You can use the commands `dump`, `append`, and `restore` to copy data between target memory and a file. The `dump` and `append` commands write data to a file, and the `restore` command reads data from a file back into the inferior's memory. Files may be in binary, Motorola S-record, Intel hex, Tektronix Hex, or Verilog Hex format; however, GDB can only append to binary files, and cannot read from Verilog Hex files.

`dump` [*format*] `memory` *filename start_addr end_addr*
`dump` [*format*] `value` *filename expr*
: Dump the contents of memory from *start_addr* to *end_addr*, or the value of *expr*, to *filename* in the given format.

  The *format* parameter may be any one of:

  `binary` Raw binary form.

  `ihex` Intel hex format.

  `srec` Motorola S-record format.

  `tekhex` Tektronix Hex format.

  `verilog` Verilog Hex format.

GDB uses the same definitions of these formats as the GNU binary utilities, like '`objdump`' and '`objcopy`'. If *format* is omitted, GDB dumps the data in raw binary form.

`append` [`binary`] `memory` *`filename start_addr end_addr`*
`append` [`binary`] `value` *`filename expr`*
: Append the contents of memory from *start_addr* to *end_addr*, or the value of *expr*, to the file *filename*, in raw binary form. (GDB can only append data to files in raw binary form.)

`restore` *`filename`* [`binary`] *`bias start end`*
: Restore the contents of file *filename* into memory. The `restore` command can automatically recognize any known BFD file format, except for raw binary. To restore a raw binary file you must specify the optional keyword `binary` after the filename.

  If *bias* is non-zero, its value will be added to the addresses contained in the file. Binary files always start at address zero, so they will be restored at address *bias*. Other bfd files have a built-in location; they will be restored at offset *bias* from that location.

  If *start* and/or *end* are non-zero, then only data between file offset *start* and file offset *end* will be restored. These offsets are relative to the addresses in the file, before the *bias* argument is applied.

10.20 How to Produce a Core File from Your Program

A *core file* or *core dump* is a file that records the memory image of a running process and its process status (register values etc.). Its primary use is post-mortem debugging of a program that crashed while it ran outside a debugger. A program that crashes automatically produces a core file, unless this feature is disabled by the user. See Section 18.1 [Files], page 277, for information on invoking GDB in the post-mortem debugging mode.

Occasionally, you may wish to produce a core file of the program you are debugging in order to preserve a snapshot of its state. GDB has a special command for that.

`generate-core-file` [*`file`*]
`gcore` [*`file`*]
: Produce a core dump of the inferior process. The optional argument *file* specifies the file name where to put the core dump. If not specified, the file name defaults to `core.`*`pid`*, where *pid* is the inferior process ID.

  Note that this command is implemented only for some systems (as of this writing, GNU/Linux, FreeBSD, Solaris, and S390).

  On GNU/Linux, this command can take into account the value of the file `/proc/`*`pid`*`/coredump_filter` when generating the core dump (see [set use-coredump-filter], page 176), and by default honors the `VM_DONTDUMP` flag for mappings where it is present in the file `/proc/`*`pid`*`/smaps` (see [set dump-excluded-mappings], page 177).

`set use-coredump-filter on`
`set use-coredump-filter off`

Enable or disable the use of the file `/proc/pid/coredump_filter` when generating core dump files. This file is used by the Linux kernel to decide what types of memory mappings will be dumped or ignored when generating a core dump file. *pid* is the process ID of a currently running process.

To make use of this feature, you have to write in the `/proc/pid/coredump_filter` file a value, in hexadecimal, which is a bit mask representing the memory mapping types. If a bit is set in the bit mask, then the memory mappings of the corresponding types will be dumped; otherwise, they will be ignored. This configuration is inherited by child processes. For more information about the bits that can be set in the `/proc/pid/coredump_filter` file, please refer to the manpage of `core(5)`.

By default, this option is `on`. If this option is turned `off`, GDB does not read the `coredump_filter` file and instead uses the same default value as the Linux kernel in order to decide which pages will be dumped in the core dump file. This value is currently `0x33`, which means that bits `0` (anonymous private mappings), `1` (anonymous shared mappings), `4` (ELF headers) and `5` (private huge pages) are active. This will cause these memory mappings to be dumped automatically.

`set dump-excluded-mappings on`
`set dump-excluded-mappings off`

If `on` is specified, GDB will dump memory mappings marked with the `VM_DONTDUMP` flag. This flag is represented in the file `/proc/pid/smaps` with the acronym `dd`.

The default value is `off`.

10.21 Character Sets

If the program you are debugging uses a different character set to represent characters and strings than the one GDB uses itself, GDB can automatically translate between the character sets for you. The character set GDB uses we call the *host character set*; the one the inferior program uses we call the *target character set*.

For example, if you are running GDB on a GNU/Linux system, which uses the ISO Latin 1 character set, but you are using GDB's remote protocol (see Chapter 20 [Remote Debugging], page 299) to debug a program running on an IBM mainframe, which uses the EBCDIC character set, then the host character set is Latin-1, and the target character set is EBCDIC. If you give GDB the command `set target-charset EBCDIC-US`, then GDB translates between EBCDIC and Latin 1 as you print character or string values, or use character and string literals in expressions.

GDB has no way to automatically recognize which character set the inferior program uses; you must tell it, using the `set target-charset` command, described below.

Here are the commands for controlling GDB's character set support:

`set target-charset` *`charset`*

Set the current target character set to *charset*. To display the list of supported target character sets, type *`set target-charset TABTAB`*.

`set host-charset` *charset*
: Set the current host character set to *charset*.

  By default, GDB uses a host character set appropriate to the system it is running on; you can override that default using the `set host-charset` command. On some systems, GDB cannot automatically determine the appropriate host character set. In this case, GDB uses '`UTF-8`'.

  GDB can only use certain character sets as its host character set. If you type *`set host-charset TABTAB`*, GDB will list the host character sets it supports.

`set charset` *charset*
: Set the current host and target character sets to *charset*. As above, if you type *`set charset TABTAB`*, GDB will list the names of the character sets that can be used for both host and target.

`show charset`
: Show the names of the current host and target character sets.

`show host-charset`
: Show the name of the current host character set.

`show target-charset`
: Show the name of the current target character set.

`set target-wide-charset` *charset*
: Set the current target's wide character set to *charset*. This is the character set used by the target's `wchar_t` type. To display the list of supported wide character sets, type *`set target-wide-charset TABTAB`*.

`show target-wide-charset`
: Show the name of the current target's wide character set.

Here is an example of GDB's character set support in action. Assume that the following source code has been placed in the file `charset-test.c`:

```
#include <stdio.h>

char ascii_hello[]
  = {72, 101, 108, 108, 111, 44, 32, 119,
     111, 114, 108, 100, 33, 10, 0};
char ibm1047_hello[]
  = {200, 133, 147, 147, 150, 107, 64, 166,
     150, 153, 147, 132, 90, 37, 0};

main ()
{
  printf ("Hello, world!\n");
}
```

In this program, `ascii_hello` and `ibm1047_hello` are arrays containing the string '`Hello, world!`' followed by a newline, encoded in the ASCII and IBM1047 character sets.

We compile the program, and invoke the debugger on it:

```
$ gcc -g charset-test.c -o charset-test
$ gdb -nw charset-test
GNU gdb 2001-12-19-cvs
Copyright 2001 Free Software Foundation, Inc.
```

```
...
(gdb)
```

We can use the `show charset` command to see what character sets GDB is currently using to interpret and display characters and strings:

```
(gdb) show charset
The current host and target character set is ‘ISO-8859-1’.
(gdb)
```

For the sake of printing this manual, let's use ASCII as our initial character set:

```
(gdb) set charset ASCII
(gdb) show charset
The current host and target character set is ‘ASCII’.
(gdb)
```

Let's assume that ASCII is indeed the correct character set for our host system — in other words, let's assume that if GDB prints characters using the ASCII character set, our terminal will display them properly. Since our current target character set is also ASCII, the contents of `ascii_hello` print legibly:

```
(gdb) print ascii_hello
$1 = 0x401698 "Hello, world!\n"
(gdb) print ascii_hello[0]
$2 = 72 'H'
(gdb)
```

GDB uses the target character set for character and string literals you use in expressions:

```
(gdb) print '+'
$3 = 43 '+'
(gdb)
```

The ASCII character set uses the number 43 to encode the '+' character.

GDB relies on the user to tell it which character set the target program uses. If we print `ibm1047_hello` while our target character set is still ASCII, we get jibberish:

```
(gdb) print ibm1047_hello
$4 = 0x4016a8 "\310\205\223\223\226k@\246\226\231\223\204Z%"
(gdb) print ibm1047_hello[0]
$5 = 200 '\310'
(gdb)
```

If we invoke the `set target-charset` followed by `TABTAB`, GDB tells us the character sets it supports:

```
(gdb) set target-charset
ASCII       EBCDIC-US   IBM1047     ISO-8859-1
(gdb) set target-charset
```

We can select IBM1047 as our target character set, and examine the program's strings again. Now the ASCII string is wrong, but GDB translates the contents of `ibm1047_hello` from the target character set, IBM1047, to the host character set, ASCII, and they display correctly:

```
(gdb) set target-charset IBM1047
(gdb) show charset
The current host character set is ‘ASCII’.
The current target character set is ‘IBM1047’.
(gdb) print ascii_hello
$6 = 0x401698 "\110\145%%?\054\040\167?\162%\144\041\012"
(gdb) print ascii_hello[0]
$7 = 72 '\110'
```

```
(gdb) print ibm1047_hello
$8 = 0x4016a8 "Hello, world!\n"
(gdb) print ibm1047_hello[0]
$9 = 200 'H'
(gdb)
```

As above, GDB uses the target character set for character and string literals you use in expressions:

```
(gdb) print '+'
$10 = 78 '+'
(gdb)
```

The IBM1047 character set uses the number 78 to encode the '+' character.

10.22 Caching Data of Targets

GDB caches data exchanged between the debugger and a target. Each cache is associated with the address space of the inferior. See Section 4.9 [Inferiors Connections and Programs], page 40, about inferior and address space. Such caching generally improves performance in remote debugging (see Chapter 20 [Remote Debugging], page 299), because it reduces the overhead of the remote protocol by bundling memory reads and writes into large chunks. Unfortunately, simply caching everything would lead to incorrect results, since GDB does not necessarily know anything about volatile values, memory-mapped I/O addresses, etc. Furthermore, in non-stop mode (see Section 5.5.2 [Non-Stop Mode], page 91) memory can be changed *while* a gdb command is executing. Therefore, by default, GDB only caches data known to be on the stack[3] or in the code segment. Other regions of memory can be explicitly marked as cacheable; see Section 10.18 [Memory Region Attributes], page 173.

`set remotecache on`
`set remotecache off`
: This option no longer does anything; it exists for compatibility with old scripts.

`show remotecache`
: Show the current state of the obsolete remotecache flag.

`set stack-cache on`
`set stack-cache off`
: Enable or disable caching of stack accesses. When `on`, use caching. By default, this option is `on`.

`show stack-cache`
: Show the current state of data caching for memory accesses.

`set code-cache on`
`set code-cache off`
: Enable or disable caching of code segment accesses. When `on`, use caching. By default, this option is `on`. This improves performance of disassembly in remote debugging.

`show code-cache`
: Show the current state of target memory cache for code segment accesses.

[3] In non-stop mode, it is moderately rare for a running thread to modify the stack of a stopped thread in a way that would interfere with a backtrace, and caching of stack reads provides a significant speed up of remote backtraces.

`info dcache` [`line`]
: Print the information about the performance of data cache of the current inferior's address space. The information displayed includes the dcache width and depth, and for each cache line, its number, address, and how many times it was referenced. This command is useful for debugging the data cache operation.

  If a line number is specified, the contents of that line will be printed in hex.

`set dcache size` *`size`*
: Set maximum number of entries in dcache (dcache depth above).

`set dcache line-size` *`line-size`*
: Set number of bytes each dcache entry caches (dcache width above). Must be a power of 2.

`show dcache size`
: Show maximum number of dcache entries. See Section 10.22 [Caching Target Data], page 180.

`show dcache line-size`
: Show default size of dcache lines.

`maint flush dcache`
: Flush the contents (if any) of the dcache. This maintainer command is useful when debugging the dcache implementation.

10.23 Search Memory

Memory can be searched for a particular sequence of bytes with the `find` command.

`find` [*`/sn`*] *`start_addr`*, *`+len`*, *`val1`* [, *`val2`*, ...]
`find` [*`/sn`*] *`start_addr`*, *`end_addr`*, *`val1`* [, *`val2`*, ...]
: Search memory for the sequence of bytes specified by *val1*, *val2*, etc. The search begins at address *start_addr* and continues for either *len* bytes or through to *end_addr* inclusive.

s and *n* are optional parameters. They may be specified in either order, apart or together.

s, search query size
: The size of each search query value.

  `b` bytes

  `h` halfwords (two bytes)

  `w` words (four bytes)

  `g` giant words (eight bytes)

  All values are interpreted in the current language. This means, for example, that if the current source language is C/C++ then searching for the string "hello" includes the trailing '\0'. The null terminator can be removed from searching by using casts, e.g.: '`{char[5]}"hello"`'.

  If the value size is not specified, it is taken from the value's type in the current language. This is useful when one wants to specify the search pattern as a

mixture of types. Note that this means, for example, that in the case of C-like languages a search for an untyped 0x42 will search for '`(int) 0x42`' which is typically four bytes.

n, maximum number of finds
: The maximum number of matches to print. The default is to print all finds.

You can use strings as search values. Quote them with double-quotes (`"`). The string value is copied into the search pattern byte by byte, regardless of the endianness of the target and the size specification.

The address of each match found is printed as well as a count of the number of matches found.

The address of the last value found is stored in convenience variable '`$_`'. A count of the number of matches is stored in '`$numfound`'.

For example, if stopped at the `printf` in this function:

```
void
hello ()
{
  static char hello[] = "hello-hello";
  static struct { char c; short s; int i; }
    __attribute__ ((packed)) mixed
    = { 'c', 0x1234, 0x87654321 };
  printf ("%s\n", hello);
}
```

you get during debugging:

```
(gdb) find &hello[0], +sizeof(hello), "hello"
0x804956d <hello.1620+6>
1 pattern found
(gdb) find &hello[0], +sizeof(hello), 'h', 'e', 'l', 'l', 'o'
0x8049567 <hello.1620>
0x804956d <hello.1620+6>
2 patterns found.
(gdb) find &hello[0], +sizeof(hello), {char[5]}"hello"
0x8049567 <hello.1620>
0x804956d <hello.1620+6>
2 patterns found.
(gdb) find /b1 &hello[0], +sizeof(hello), 'h', 0x65, 'l'
0x8049567 <hello.1620>
1 pattern found
(gdb) find &mixed, +sizeof(mixed), (char) 'c', (short) 0x1234, (int) 0x87654321
0x8049560 <mixed.1625>
1 pattern found
(gdb) print $numfound
$1 = 1
(gdb) print $_
$2 = (void *) 0x8049560
```

10.24 Value Sizes

Whenever GDB prints a value memory will be allocated within GDB to hold the contents of the value. It is possible in some languages with dynamic typing systems, that an invalid program may indicate a value that is incorrectly large, this in turn may cause GDB to try and allocate an overly large amount of memory.

`set max-value-size` *bytes*
`set max-value-size unlimited`

Set the maximum size of memory that GDB will allocate for the contents of a value to *bytes*, trying to display a value that requires more memory than that will result in an error.

Setting this variable does not effect values that have already been allocated within GDB, only future allocations.

There's a minimum size that `max-value-size` can be set to in order that GDB can still operate correctly, this minimum is currently 16 bytes.

The limit applies to the results of some subexpressions as well as to complete expressions. For example, an expression denoting a simple integer component, such as `x.y.z`, may fail if the size of *x.y* is dynamic and exceeds *bytes*. On the other hand, GDB is sometimes clever; the expression `A[i]`, where *A* is an array variable with non-constant size, will generally succeed regardless of the bounds on *A*, as long as the component size is less than *bytes*.

The default value of `max-value-size` is currently 64k.

`show max-value-size`

Show the maximum size of memory, in bytes, that GDB will allocate for the contents of a value.

11 Debugging Optimized Code

Almost all compilers support optimization. With optimization disabled, the compiler generates assembly code that corresponds directly to your source code, in a simplistic way. As the compiler applies more powerful optimizations, the generated assembly code diverges from your original source code. With help from debugging information generated by the compiler, GDB can map from the running program back to constructs from your original source.

GDB is more accurate with optimization disabled. If you can recompile without optimization, it is easier to follow the progress of your program during debugging. But, there are many cases where you may need to debug an optimized version.

When you debug a program compiled with '`-g -O`', remember that the optimizer has rearranged your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example: if you define a variable, but never use it, GDB never sees that variable—because the compiler optimizes it out of existence.

Some things do not work as well with '`-g -O`' as with just '`-g`', particularly on machines with instruction scheduling. If in doubt, recompile with '`-g`' alone, and if this fixes the problem, please report it to us as a bug (including a test case!). See Section 10.3 [Variables], page 138, for more information about debugging optimized code.

11.1 Inline Functions

Inlining is an optimization that inserts a copy of the function body directly at each call site, instead of jumping to a shared routine. GDB displays inlined functions just like non-inlined functions. They appear in backtraces. You can view their arguments and local variables, step into them with `step`, skip them with `next`, and escape from them with `finish`. You can check whether a function was inlined by using the `info frame` command.

For GDB to support inlined functions, the compiler must record information about inlining in the debug information — GCC using the DWARF 2 format does this, and several other compilers do also. GDB only supports inlined functions when using DWARF 2. Versions of GCC before 4.1 do not emit two required attributes ('`DW_AT_call_file`' and '`DW_AT_call_line`'); GDB does not display inlined function calls with earlier versions of GCC. It instead displays the arguments and local variables of inlined functions as local variables in the caller.

The body of an inlined function is directly included at its call site; unlike a non-inlined function, there are no instructions devoted to the call. GDB still pretends that the call site and the start of the inlined function are different instructions. Stepping to the call site shows the call site, and then stepping again shows the first line of the inlined function, even though no additional instructions are executed.

This makes source-level debugging much clearer; you can see both the context of the call and then the effect of the call. Only stepping by a single instruction using `stepi` or `nexti` does not do this; single instruction steps always show the inlined body.

There are some ways that GDB does not pretend that inlined function calls are the same as normal calls:

- Setting breakpoints at the call site of an inlined function may not work, because the call site does not contain any code. GDB may incorrectly move the breakpoint to the next line of the enclosing function, after the call. This limitation will be removed in a future version of GDB; until then, set a breakpoint on an earlier line or inside the inlined function instead.
- GDB cannot locate the return value of inlined calls after using the `finish` command. This is a limitation of compiler-generated debugging information; after `finish`, you can step to the next line and print a variable where your program stored the return value.

11.2 Tail Call Frames

Function `B` can call function `C` in its very last statement. In unoptimized compilation the call of `C` is immediately followed by return instruction at the end of `B` code. Optimizing compiler may replace the call and return in function `B` into one jump to function `C` instead. Such use of a jump instruction is called *tail call*.

During execution of function `C`, there will be no indication in the function call stack frames that it was tail-called from `B`. If function `A` regularly calls function `B` which tail-calls function `C`, then GDB will see `A` as the caller of `C`. However, in some cases GDB can determine that `C` was tail-called from `B`, and it will then create fictitious call frame for that, with the return address set up as if `B` called `C` normally.

This functionality is currently supported only by DWARF 2 debugging format and the compiler has to produce '`DW_TAG_call_site`' tags. With GCC, you need to specify `-O -g` during compilation, to get this information.

`info frame` command (see Section 8.4 [Frame Info], page 113) will indicate the tail call frame kind by text `tail call frame` such as in this sample GDB output:

```
(gdb) x/i $pc - 2
   0x40066b <b(int, double)+11>: jmp 0x400640 <c(int, double)>
(gdb) info frame
Stack level 1, frame at 0x7fffffffda30:
 rip = 0x40066d in b (amd64-entry-value.cc:59); saved rip 0x4004c5
 tail call frame, caller of frame at 0x7fffffffda30
 source language c++.
 Arglist at unknown address.
 Locals at unknown address, Previous frame's sp is 0x7fffffffda30
```

The detection of all the possible code path executions can find them ambiguous. There is no execution history stored (possible Chapter 6 [Reverse Execution], page 97, is never used for this purpose) and the last known caller could have reached the known callee by multiple different jump sequences. In such case GDB still tries to show at least all the unambiguous top tail callers and all the unambiguous bottom tail calees, if any.

`set debug entry-values`

When set to on, enables printing of analysis messages for both frame argument values at function entry and tail calls. It will show all the possible valid tail calls code paths it has considered. It will also print the intersection of them with the final unambiguous (possibly partial or even empty) code path result.

`show debug entry-values`
: Show the current state of analysis messages printing for both frame argument values at function entry and tail calls.

The analysis messages for tail calls can for example show why the virtual tail call frame for function `c` has not been recognized (due to the indirect reference by variable `x`):

```
static void __attribute__((noinline, noclone)) c (void);
void (*x) (void) = c;
static void __attribute__((noinline, noclone)) a (void) { x++; }
static void __attribute__((noinline, noclone)) c (void) { a (); }
int main (void) { x (); return 0; }

Breakpoint 1, DW_OP_entry_value resolving cannot find
DW_TAG_call_site 0x40039a in main
a () at t.c:3
3 static void __attribute__((noinline, noclone)) a (void) { x++; }
(gdb) bt
#0  a () at t.c:3
#1  0x000000000040039a in main () at t.c:5
```

Another possibility is an ambiguous virtual tail call frames resolution:

```
int i;
static void __attribute__((noinline, noclone)) f (void) { i++; }
static void __attribute__((noinline, noclone)) e (void) { f (); }
static void __attribute__((noinline, noclone)) d (void) { f (); }
static void __attribute__((noinline, noclone)) c (void) { d (); }
static void __attribute__((noinline, noclone)) b (void)
{ if (i) c (); else e (); }
static void __attribute__((noinline, noclone)) a (void) { b (); }
int main (void) { a (); return 0; }

tailcall: initial: 0x4004d2(a) 0x4004ce(b) 0x4004b2(c) 0x4004a2(d)
tailcall: compare: 0x4004d2(a) 0x4004cc(b) 0x400492(e)
tailcall: reduced: 0x4004d2(a) |
(gdb) bt
#0  f () at t.c:2
#1  0x00000000004004d2 in a () at t.c:8
#2  0x0000000000400395 in main () at t.c:9
```

Frames #0 and #2 are real, #1 is a virtual tail call frame. The code can have possible execution paths `main->a->b->c->d->f` or `main->a->b->e->f`, GDB cannot find which one from the inferior state.

`initial:` state shows some random possible calling sequence GDB has found. It then finds another possible calling sequence - that one is prefixed by `compare:`. The non-ambiguous intersection of these two is printed as the `reduced:` calling sequence. That one could have many further `compare:` and `reduced:` statements as long as there remain any non-ambiguous sequence entries.

For the frame of function `b` in both cases there are different possible `$pc` values (`0x4004cc` or `0x4004ce`), therefore this frame is also ambiguous. The only non-ambiguous frame is the one for function `a`, therefore this one is displayed to the user while the ambiguous frames are omitted.

There can be also reasons why printing of frame argument values at function entry may fail:

```
int v;
```

```
static void __attribute__((noinline, noclone)) c (int i) { v++; }
static void __attribute__((noinline, noclone)) a (int i);
static void __attribute__((noinline, noclone)) b (int i) { a (i); }
static void __attribute__((noinline, noclone)) a (int i)
{ if (i) b (i - 1); else c (0); }
int main (void) { a (5); return 0; }

(gdb) bt
#0  c (i=i@entry=0) at t.c:2
#1  0x0000000000400428 in a (DW_OP_entry_value resolving has found
function "a" at 0x400420 can call itself via tail calls
i=<optimized out>) at t.c:6
#2  0x000000000040036e in main () at t.c:7
```

GDB cannot find out from the inferior state if and how many times did function `a` call itself (via function `b`) as these calls would be tail calls. Such tail calls would modify the `i` variable, therefore GDB cannot be sure the value it knows would be right - GDB prints `<optimized out>` instead.

12 C Preprocessor Macros

Some languages, such as C and C++, provide a way to define and invoke "preprocessor macros" which expand into strings of tokens. GDB can evaluate expressions containing macro invocations, show the result of macro expansion, and show a macro's definition, including where it was defined.

You may need to compile your program specially to provide GDB with information about preprocessor macros. Most compilers do not include macros in their debugging information, even when you compile with the `-g` flag. See Section 4.1 [Compilation], page 31.

A program may define a macro at one point, remove that definition later, and then provide a different definition after that. Thus, at different points in the program, a macro may have different definitions, or have no definition at all. If there is a current stack frame, GDB uses the macros in scope at that frame's source code line. Otherwise, GDB uses the macros in scope at the current listing location; see Section 9.1 [List], page 119.

Whenever GDB evaluates an expression, it always expands any macro invocations present in the expression. GDB also provides the following commands for working with macros explicitly.

`macro expand` *`expression`*
`macro exp` *`expression`*
: Show the results of expanding all preprocessor macro invocations in *expression*. Since GDB simply expands macros, but does not parse the result, *expression* need not be a valid expression; it can be any string of tokens.

`macro expand-once` *`expression`*
`macro exp1` *`expression`*
: *(This command is not yet implemented.)* Show the results of expanding those preprocessor macro invocations that appear explicitly in *expression*. Macro invocations appearing in that expansion are left unchanged. This command allows you to see the effect of a particular macro more clearly, without being confused by further expansions. Since GDB simply expands macros, but does not parse the result, *expression* need not be a valid expression; it can be any string of tokens.

`info macro [-a|-all] [--]` *`macro`*
: Show the current definition or all definitions of the named *macro*, and describe the source location or compiler command-line where that definition was established. The optional double dash is to signify the end of argument processing and the beginning of *macro* for non C-like macros where the macro may begin with a hyphen.

`info macros` *`location`*
: Show all macro definitions that are in effect at the location specified by *location*, and describe the source location or compiler command-line where those definitions were established.

`macro define` *`macro replacement-list`*
`macro define` *`macro(arglist) replacement-list`*
: Introduce a definition for a preprocessor macro named *macro*, invocations of which are replaced by the tokens given in *replacement-list*. The first form of

this command defines an "object-like" macro, which takes no arguments; the second form defines a "function-like" macro, which takes the arguments given in *arglist*.

A definition introduced by this command is in scope in every expression evaluated in GDB, until it is removed with the `macro undef` command, described below. The definition overrides all definitions for *macro* present in the program being debugged, as well as any previous user-supplied definition.

`macro undef` *macro*
: Remove any user-supplied definition for the macro named *macro*. This command only affects definitions provided with the `macro define` command, described above; it cannot remove definitions present in the program being debugged.

`macro list`
: List all the macros defined using the `macro define` command.

Here is a transcript showing the above commands in action. First, we show our source files:

```
$ cat sample.c
#include <stdio.h>
#include "sample.h"

#define M 42
#define ADD(x) (M + x)

main ()
{
#define N 28
  printf ("Hello, world!\n");
#undef N
  printf ("We're so creative.\n");
#define N 1729
  printf ("Goodbye, world!\n");
}
$ cat sample.h
#define Q <
$
```

Now, we compile the program using the GNU C compiler, GCC. We pass the `-gdwarf-2`[1] *and* `-g3` flags to ensure the compiler includes information about preprocessor macros in the debugging information.

```
$ gcc -gdwarf-2 -g3 sample.c -o sample
$
```

Now, we start GDB on our sample program:

```
$ gdb -nw sample
GNU gdb 2002-05-06-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, ...
(gdb)
```

[1] This is the minimum. Recent versions of GCC support `-gdwarf-3` and `-gdwarf-4`; we recommend always choosing the most recent version of DWARF.

We can expand macros and examine their definitions, even when the program is not running. GDB uses the current listing position to decide which macro definitions are in scope:

```
(gdb) list main
3
4       #define M 42
5       #define ADD(x) (M + x)
6
7       main ()
8       {
9       #define N 28
10        printf ("Hello, world!\n");
11      #undef N
12        printf ("We're so creative.\n");
(gdb) info macro ADD
Defined at /home/jimb/gdb/macros/play/sample.c:5
#define ADD(x) (M + x)
(gdb) info macro Q
Defined at /home/jimb/gdb/macros/play/sample.h:1
  included at /home/jimb/gdb/macros/play/sample.c:2
#define Q <
(gdb) macro expand ADD(1)
expands to: (42 + 1)
(gdb) macro expand-once ADD(1)
expands to: once (M + 1)
(gdb)
```

In the example above, note that `macro expand-once` expands only the macro invocation explicit in the original text — the invocation of `ADD` — but does not expand the invocation of the macro `M`, which was introduced by `ADD`.

Once the program is running, GDB uses the macro definitions in force at the source line of the current stack frame:

```
(gdb) break main
Breakpoint 1 at 0x8048370: file sample.c, line 10.
(gdb) run
Starting program: /home/jimb/gdb/macros/play/sample

Breakpoint 1, main () at sample.c:10
10        printf ("Hello, world!\n");
(gdb)
```

At line 10, the definition of the macro `N` at line 9 is in force:

```
(gdb) info macro N
Defined at /home/jimb/gdb/macros/play/sample.c:9
#define N 28
(gdb) macro expand N Q M
expands to: 28 < 42
(gdb) print N Q M
$1 = 1
(gdb)
```

As we step over directives that remove `N`'s definition, and then give it a new definition, GDB finds the definition (or lack thereof) in force at each point:

```
(gdb) next
Hello, world!
12        printf ("We're so creative.\n");
(gdb) info macro N
```

```
The symbol 'N' has no definition as a C/C++ preprocessor macro
at /home/jimb/gdb/macros/play/sample.c:12
(gdb) next
We're so creative.
14        printf ("Goodbye, world!\n");
(gdb) info macro N
Defined at /home/jimb/gdb/macros/play/sample.c:13
#define N 1729
(gdb) macro expand N Q M
expands to: 1729 < 42
(gdb) print N Q M
$2 = 0
(gdb)
```

In addition to source files, macros can be defined on the compilation command line using the `-D`*`name`*`=`*`value`* syntax. For macros defined in such a way, GDB displays the location of their definition as line zero of the source file submitted to the compiler.

```
(gdb) info macro __STDC__
Defined at /home/jimb/gdb/macros/play/sample.c:0
-D__STDC__=1
(gdb)
```

13 Tracepoints

In some applications, it is not feasible for the debugger to interrupt the program's execution long enough for the developer to learn anything helpful about its behavior. If the program's correctness depends on its real-time behavior, delays introduced by a debugger might cause the program to change its behavior drastically, or perhaps fail, even when the code itself is correct. It is useful to be able to observe the program's behavior without interrupting it.

Using GDB's `trace` and `collect` commands, you can specify locations in the program, called *tracepoints*, and arbitrary expressions to evaluate when those tracepoints are reached. Later, using the `tfind` command, you can examine the values those expressions had when the program hit the tracepoints. The expressions may also denote objects in memory—structures or arrays, for example—whose values GDB should record; while visiting a particular tracepoint, you may inspect those objects as if they were in memory at that moment. However, because GDB records these values without interacting with you, it can do so quickly and unobtrusively, hopefully not disturbing the program's behavior.

The tracepoint facility is currently available only for remote targets. See Chapter 19 [Targets], page 295. In addition, your remote target must know how to collect trace data. This functionality is implemented in the remote stub; however, none of the stubs distributed with GDB support tracepoints as of this writing. The format of the remote packets used to implement tracepoints are described in Section E.6 [Tracepoint Packets], page 725.

It is also possible to get trace data from a file, in a manner reminiscent of corefiles; you specify the filename, and use `tfind` to search through the file. See Section 13.4 [Trace Files], page 209, for more details.

This chapter describes the tracepoint commands and features.

13.1 Commands to Set Tracepoints

Before running such a *trace experiment*, an arbitrary number of tracepoints can be set. A tracepoint is actually a special type of breakpoint (see Section 5.1.1 [Set Breaks], page 56), so you can manipulate it using standard breakpoint commands. For instance, as with breakpoints, tracepoint numbers are successive integers starting from one, and many of the commands associated with tracepoints take the tracepoint number as their argument, to identify which tracepoint to work on.

For each tracepoint, you can specify, in advance, some arbitrary set of data that you want the target to collect in the trace buffer when it hits that tracepoint. The collected data can include registers, local variables, or global data. Later, you can use GDB commands to examine the values these data had at the time the tracepoint was hit.

Tracepoints do not support every breakpoint feature. Ignore counts on tracepoints have no effect, and tracepoints cannot run GDB commands when they are hit. Tracepoints may not be thread-specific either.

Some targets may support *fast tracepoints*, which are inserted in a different way (such as with a jump instead of a trap), that is faster but possibly restricted in where they may be installed.

Regular and fast tracepoints are dynamic tracing facilities, meaning that they can be used to insert tracepoints at (almost) any location in the target. Some targets may also support controlling *static tracepoints* from GDB. With static tracing, a set of instrumentation

points, also known as *markers*, are embedded in the target program, and can be activated or deactivated by name or address. These are usually placed at locations which facilitate investigating what the target is actually doing. GDB's support for static tracing includes being able to list instrumentation points, and attach them with GDB defined high level tracepoints that expose the whole range of convenience of GDB's tracepoints support. Namely, support for collecting registers values and values of global or local (to the instrumentation point) variables; tracepoint conditions and trace state variables. The act of installing a GDB static tracepoint on an instrumentation point, or marker, is referred to as *probing* a static tracepoint marker.

`gdbserver` supports tracepoints on some target systems. See Section 20.3 [Tracepoints support in gdbserver], page 304.

This section describes commands to set tracepoints and associated conditions and actions.

13.1.1 Create and Delete Tracepoints

`trace` *location*

The `trace` command is very similar to the `break` command. Its argument *location* can be any valid location. See Section 9.2 [Specify Location], page 120. The `trace` command defines a tracepoint, which is a point in the target program where the debugger will briefly stop, collect some data, and then allow the program to continue. Setting a tracepoint or changing its actions takes effect immediately if the remote stub supports the '`InstallInTrace`' feature (see [install tracepoint in tracing], page 716). If remote stub doesn't support the '`InstallInTrace`' feature, all these changes don't take effect until the next `tstart` command, and once a trace experiment is running, further changes will not have any effect until the next trace experiment starts. In addition, GDB supports *pending tracepoints*—tracepoints whose address is not yet resolved. (This is similar to pending breakpoints.) Pending tracepoints are not downloaded to the target and not installed until they are resolved. The resolution of pending tracepoints requires GDB support—when debugging with the remote target, and GDB disconnects from the remote stub (see [disconnected tracing], page 202), pending tracepoints can not be resolved (and downloaded to the remote stub) while GDB is disconnected.

Here are some examples of using the `trace` command:

```
(gdb) trace foo.c:121     // a source file and line number

(gdb) trace +2            // 2 lines forward

(gdb) trace my_function   // first source line of function

(gdb) trace *my_function // EXACT start address of function

(gdb) trace *0x2117c4     // an address
```

You can abbreviate `trace` as `tr`.

`trace` *location* `if` *cond*

Set a tracepoint with condition *cond*; evaluate the expression *cond* each time the tracepoint is reached, and collect data only if the value is nonzero—that is,

if *cond* evaluates as true. See Section 13.1.4 [Tracepoint Conditions], page 197, for more information on tracepoint conditions.

`ftrace` *location* [`if` *cond*]

The `ftrace` command sets a fast tracepoint. For targets that support them, fast tracepoints will use a more efficient but possibly less general technique to trigger data collection, such as a jump instruction instead of a trap, or some sort of hardware support. It may not be possible to create a fast tracepoint at the desired location, in which case the command will exit with an explanatory message.

GDB handles arguments to `ftrace` exactly as for `trace`.

On 32-bit x86-architecture systems, fast tracepoints normally need to be placed at an instruction that is 5 bytes or longer, but can be placed at 4-byte instructions if the low 64K of memory of the target program is available to install trampolines. Some Unix-type systems, such as GNU/Linux, exclude low addresses from the program's address space; but for instance with the Linux kernel it is possible to let GDB use this area by doing a `sysctl` command to set the `mmap_min_addr` kernel parameter, as in

```
sudo sysctl -w vm.mmap_min_addr=32768
```

which sets the low address to 32K, which leaves plenty of room for trampolines. The minimum address should be set to a page boundary.

`strace` *location* [`if` *cond*]

The `strace` command sets a static tracepoint. For targets that support it, setting a static tracepoint probes a static instrumentation point, or marker, found at *location*. It may not be possible to set a static tracepoint at the desired location, in which case the command will exit with an explanatory message.

GDB handles arguments to `strace` exactly as for `trace`, with the addition that the user can also specify `-m` *marker* as *location*. This probes the marker identified by the *marker* string identifier. This identifier depends on the static tracepoint backend library your program is using. You can find all the marker identifiers in the 'ID' field of the `info static-tracepoint-markers` command output. See Section 13.1.8 [Listing Static Tracepoint Markers], page 201. For example, in the following small program using the UST tracing engine:

```
main ()
{
  trace_mark(ust, bar33, "str %s", "FOOBAZ");
}
```

the marker id is composed of joining the first two arguments to the `trace_mark` call with a slash, which translates to:

```
(gdb) info static-tracepoint-markers
Cnt Enb ID         Address            What
1   n   ust/bar33  0x0000000000400ddc in main at stexample.c:22
         Data: "str %s"
[etc...]
```

so you may probe the marker above with:

```
(gdb) strace -m ust/bar33
```

Static tracepoints accept an extra collect action — `collect $_sdata`. This collects arbitrary user data passed in the probe point call to the tracing library. In the UST example above, you'll see that the third argument to `trace_mark` is a printf-like format string. The user data is then the result of running that formatting string against the following arguments. Note that `info static-tracepoint-markers` command output lists that format string in the '`Data:`' field.

You can inspect this data when analyzing the trace buffer, by printing the $_sdata variable like any other variable available to GDB. See Section 13.1.6 [Tracepoint Action Lists], page 198.

The convenience variable `$tpnum` records the tracepoint number of the most recently set tracepoint.

`delete tracepoint` [*num*]

Permanently delete one or more tracepoints. With no argument, the default is to delete all tracepoints. Note that the regular `delete` command can remove tracepoints also.

Examples:

```
(gdb) delete trace 1 2 3 // remove three tracepoints

(gdb) delete trace       // remove all tracepoints
```

You can abbreviate this command as `del tr`.

13.1.2 Enable and Disable Tracepoints

These commands are deprecated; they are equivalent to plain `disable` and `enable`.

`disable tracepoint` [*num*]

Disable tracepoint *num*, or all tracepoints if no argument *num* is given. A disabled tracepoint will have no effect during a trace experiment, but it is not forgotten. You can re-enable a disabled tracepoint using the `enable tracepoint` command. If the command is issued during a trace experiment and the debug target has support for disabling tracepoints during a trace experiment, then the change will be effective immediately. Otherwise, it will be applied to the next trace experiment.

`enable tracepoint` [*num*]

Enable tracepoint *num*, or all tracepoints. If this command is issued during a trace experiment and the debug target supports enabling tracepoints during a trace experiment, then the enabled tracepoints will become effective immediately. Otherwise, they will become effective the next time a trace experiment is run.

13.1.3 Tracepoint Passcounts

`passcount` [*n* [*num*]]

Set the *passcount* of a tracepoint. The passcount is a way to automatically stop a trace experiment. If a tracepoint's passcount is *n*, then the trace experiment will be automatically stopped on the *n*'th time that tracepoint is hit. If the tracepoint number *num* is not specified, the `passcount` command sets the

passcount of the most recently defined tracepoint. If no passcount is given, the trace experiment will run until stopped explicitly by the user.

Examples:

```
(gdb) passcount 5 2 // Stop on the 5th execution of
                    // tracepoint 2

(gdb) passcount 12  // Stop on the 12th execution of the
                    // most recently defined tracepoint.
(gdb) trace foo
(gdb) pass 3
(gdb) trace bar
(gdb) pass 2
(gdb) trace baz
(gdb) pass 1        // Stop tracing when foo has been
                    // executed 3 times OR when bar has
                    // been executed 2 times
                    // OR when baz has been executed 1 time.
```

13.1.4 Tracepoint Conditions

The simplest sort of tracepoint collects data every time your program reaches a specified place. You can also specify a *condition* for a tracepoint. A condition is just a Boolean expression in your programming language (see Section 10.1 [Expressions], page 137). A tracepoint with a condition evaluates the expression each time your program reaches it, and data collection happens only if the condition is true.

Tracepoint conditions can be specified when a tracepoint is set, by using '`if`' in the arguments to the `trace` command. See Section 13.1.1 [Setting Tracepoints], page 194. They can also be set or changed at any time with the `condition` command, just as with breakpoints.

Unlike breakpoint conditions, GDB does not actually evaluate the conditional expression itself. Instead, GDB encodes the expression into an agent expression (see Appendix F [Agent Expressions], page 757) suitable for execution on the target, independently of GDB. Global variables become raw memory locations, locals become stack accesses, and so forth.

For instance, suppose you have a function that is usually called frequently, but should not be called after an error has occurred. You could use the following tracepoint command to collect data about calls of that function that happen while the error code is propagating through the program; an unconditional tracepoint could end up collecting thousands of useless trace frames that you would have to search through.

```
(gdb) trace normal_operation if errcode > 0
```

13.1.5 Trace State Variables

A *trace state variable* is a special type of variable that is created and managed by target-side code. The syntax is the same as that for GDB's convenience variables (a string prefixed with "$"), but they are stored on the target. They must be created explicitly, using a `tvariable` command. They are always 64-bit signed integers.

Trace state variables are remembered by GDB, and downloaded to the target along with tracepoint information when the trace experiment starts. There are no intrinsic limits on the number of trace state variables, beyond memory limitations of the target.

Although trace state variables are managed by the target, you can use them in print commands and expressions as if they were convenience variables; GDB will get the current value from the target while the trace experiment is running. Trace state variables share the same namespace as other "$" variables, which means that you cannot have trace state variables with names like `$23` or `$pc`, nor can you have a trace state variable and a convenience variable with the same name.

`tvariable $`*`name`* `[ =` *`expression`* `]`

The `tvariable` command creates a new trace state variable named `$`*`name`*, and optionally gives it an initial value of *expression*. The *expression* is evaluated when this command is entered; the result will be converted to an integer if possible, otherwise GDB will report an error. A subsequent `tvariable` command specifying the same name does not create a variable, but instead assigns the supplied initial value to the existing variable of that name, overwriting any previous initial value. The default initial value is 0.

`info tvariables`

List all the trace state variables along with their initial values. Their current values may also be displayed, if the trace experiment is currently running.

`delete tvariable` [`$`*`name`* `...`]

Delete the given trace state variables, or all of them if no arguments are specified.

13.1.6 Tracepoint Action Lists

`actions` [*`num`*]

This command will prompt for a list of actions to be taken when the tracepoint is hit. If the tracepoint number *num* is not specified, this command sets the actions for the one that was most recently defined (so that you can define a tracepoint and then say `actions` without bothering about its number). You specify the actions themselves on the following lines, one action at a time, and terminate the actions list with a line containing just `end`. So far, the only defined actions are `collect`, `teval`, and `while-stepping`.

`actions` is actually equivalent to `commands` (see Section 5.1.7 [Breakpoint Command Lists], page 74), except that only the defined actions are allowed; any other GDB command is rejected.

To remove all actions from a tracepoint, type '`actions` *`num`*' and follow it immediately with '`end`'.

```
(gdb) collect data // collect some data

(gdb) while-stepping 5 // single-step 5 times, collect data

(gdb) end              // signals the end of actions.
```

In the following example, the action list begins with `collect` commands indicating the things to be collected when the tracepoint is hit. Then, in order to single-step and collect additional data following the tracepoint, a `while-stepping` command is used, followed by the list of things to be collected after each step in a sequence of single steps. The `while-stepping` command is terminated by

its own separate `end` command. Lastly, the action list is terminated by an `end` command.

```
(gdb) trace foo
(gdb) actions
Enter actions for tracepoint 1, one per line:
> collect bar,baz
> collect $regs
> while-stepping 12
  > collect $pc, arr[i]
  > end
end
```

`collect`[`/`*mods*] *expr1*`,` *expr2*`, ...`

Collect values of the given expressions when the tracepoint is hit. This command accepts a comma-separated list of any valid expressions. In addition to global, static, or local variables, the following special arguments are supported:

`$regs` Collect all registers.

`$args` Collect all function arguments.

`$locals` Collect all local variables.

`$_ret` Collect the return address. This is helpful if you want to see more of a backtrace.

Note: The return address location can not always be reliably determined up front, and the wrong address / registers may end up collected instead. On some architectures the reliability is higher for tracepoints at function entry, while on others it's the opposite. When this happens, backtracing will stop because the return address is found unavailable (unless another collect rule happened to match it).

`$_probe_argc`

Collects the number of arguments from the static probe at which the tracepoint is located. See Section 5.1.10 [Static Probe Points], page 77.

`$_probe_arg`*n*

n is an integer between 0 and 11. Collects the *n*th argument from the static probe at which the tracepoint is located. See Section 5.1.10 [Static Probe Points], page 77.

`$_sdata` Collect static tracepoint marker specific data. Only available for static tracepoints. See Section 13.1.6 [Tracepoint Action Lists], page 198. On the UST static tracepoints library backend, an instrumentation point resembles a `printf` function call. The tracing library is able to collect user specified data formatted to a character string using the format provided by the programmer that instrumented the program. Other backends have similar mechanisms. Here's an example of a UST marker call:

```
const char master_name[] = "$your_name";
trace_mark(channel1, marker1, "hello %s", master_name)
```

In this case, collecting `$_sdata` collects the string '`hello $yourname`'. When analyzing the trace buffer, you can inspect '`$_sdata`' like any other variable available to GDB.

You can give several consecutive `collect` commands, each one with a single argument, or one `collect` command with several arguments separated by commas; the effect is the same.

The optional *mods* changes the usual handling of the arguments. `s` requests that pointers to chars be handled as strings, in particular collecting the contents of the memory being pointed at, up to the first zero. The upper bound is by default the value of the `print elements` variable; if `s` is followed by a decimal number, that is the upper bound instead. So for instance '`collect/s25 mystr`' collects as many as 25 characters at '`mystr`'.

The command `info scope` (see Chapter 16 [Symbols], page 251) is particularly useful for figuring out what data to collect.

`teval` *expr1, expr2, ...*

Evaluate the given expressions when the tracepoint is hit. This command accepts a comma-separated list of expressions. The results are discarded, so this is mainly useful for assigning values to trace state variables (see Section 13.1.5 [Trace State Variables], page 197) without adding those values to the trace buffer, as would be the case if the `collect` action were used.

`while-stepping` *n*

Perform *n* single-step instruction traces after the tracepoint, collecting new data after each step. The `while-stepping` command is followed by the list of what to collect while stepping (followed by its own `end` command):

```
> while-stepping 12
  > collect $regs, myglobal
  > end
>
```

Note that `$pc` is not automatically collected by `while-stepping`; you need to explicitly collect that register if you need it. You may abbreviate `while-stepping` as `ws` or `stepping`.

`set default-collect` *expr1, expr2, ...*

This variable is a list of expressions to collect at each tracepoint hit. It is effectively an additional `collect` action prepended to every tracepoint action list. The expressions are parsed individually for each tracepoint, so for instance a variable named `xyz` may be interpreted as a global for one tracepoint, and a local for another, as appropriate to the tracepoint's location.

`show default-collect`

Show the list of expressions that are collected by default at each tracepoint hit.

13.1.7 Listing Tracepoints

`info tracepoints` [*num...*]

Display information about the tracepoint *num*. If you don't specify a tracepoint number, displays information about all the tracepoints defined so far. The

format is similar to that used for `info breakpoints`; in fact, `info tracepoints` is the same command, simply restricting itself to tracepoints.

A tracepoint's listing may include additional information specific to tracing:

- its passcount as given by the `passcount n` command
- the state about installed on target of each location

```
(gdb) info trace
Num     Type           Disp Enb Address    What
1       tracepoint     keep y   0x0804ab57 in foo() at main.cxx:7
        while-stepping 20
          collect globfoo, $regs
        end
        collect globfoo2
        end
        pass count 1200
2       tracepoint     keep y   <MULTIPLE>
        collect $eip
2.1                    y     0x0804859c in func4 at change-loc.h:35
        installed on target
2.2                    y     0xb7ffc480 in func4 at change-loc.h:35
        installed on target
2.3                    y     <PENDING>  set_tracepoint
3       tracepoint     keep y   0x080485b1 in foo at change-loc.c:29
        not installed on target
(gdb)
```

This command can be abbreviated `info tp`.

13.1.8 Listing Static Tracepoint Markers

`info static-tracepoint-markers`

Display information about all static tracepoint markers defined in the program.

For each marker, the following columns are printed:

Count An incrementing counter, output to help readability. This is not a stable identifier.

ID The marker ID, as reported by the target.

Enabled or Disabled
Probed markers are tagged with '`y`'. '`n`' identifies marks that are not enabled.

Address Where the marker is in your program, as a memory address.

What Where the marker is in the source for your program, as a file and line number. If the debug information included in the program does not allow GDB to locate the source of the marker, this column will be left blank.

In addition, the following information may be printed for each marker:

Data User data passed to the tracing library by the marker call. In the UST backend, this is the format string passed as argument to the marker call.

Static tracepoints probing the marker
: The list of static tracepoints attached to the marker.

```
(gdb) info static-tracepoint-markers
Cnt ID         Enb Address            What
1   ust/bar2   y   0x0000000000400e1a in main at stexample.c:25
     Data: number1 %d number2 %d
     Probed by static tracepoints: #2
2   ust/bar33  n   0x0000000000400c87 in main at stexample.c:24
     Data: str %s
(gdb)
```

13.1.9 Starting and Stopping Trace Experiments

`tstart`
: This command starts the trace experiment, and begins collecting data. It has the side effect of discarding all the data collected in the trace buffer during the previous trace experiment. If any arguments are supplied, they are taken as a note and stored with the trace experiment's state. The notes may be arbitrary text, and are especially useful with disconnected tracing in a multi-user context; the notes can explain what the trace is doing, supply user contact information, and so forth.

`tstop`
: This command stops the trace experiment. If any arguments are supplied, they are recorded with the experiment as a note. This is useful if you are stopping a trace started by someone else, for instance if the trace is interfering with the system's behavior and needs to be stopped quickly.

  Note: a trace experiment and data collection may stop automatically if any tracepoint's passcount is reached (see Section 13.1.3 [Tracepoint Passcounts], page 196), or if the trace buffer becomes full.

`tstatus`
: This command displays the status of the current trace data collection.

Here is an example of the commands we described so far:

```
(gdb) trace gdb_c_test
(gdb) actions
Enter actions for tracepoint #1, one per line.
> collect $regs,$locals,$args
> while-stepping 11
  > collect $regs
  > end
> end
(gdb) tstart
[time passes ...]
(gdb) tstop
```

You can choose to continue running the trace experiment even if GDB disconnects from the target, voluntarily or involuntarily. For commands such as `detach`, the debugger will ask what you want to do with the trace. But for unexpected terminations (GDB crash, network outage), it would be unfortunate to lose hard-won trace data, so the variable `disconnected-tracing` lets you decide whether the trace should continue running without GDB.

`set disconnected-tracing on`
`set disconnected-tracing off`
: Choose whether a tracing run should continue to run if GDB has disconnected from the target. Note that `detach` or `quit` will ask you directly what to do about a running trace no matter what this variable's setting, so the variable is mainly useful for handling unexpected situations, such as loss of the network.

`show disconnected-tracing`
: Show the current choice for disconnected tracing.

When you reconnect to the target, the trace experiment may or may not still be running; it might have filled the trace buffer in the meantime, or stopped for one of the other reasons. If it is running, it will continue after reconnection.

Upon reconnection, the target will upload information about the tracepoints in effect. GDB will then compare that information to the set of tracepoints currently defined, and attempt to match them up, allowing for the possibility that the numbers may have changed due to creation and deletion in the meantime. If one of the target's tracepoints does not match any in GDB, the debugger will create a new tracepoint, so that you have a number with which to specify that tracepoint. This matching-up process is necessarily heuristic, and it may result in useless tracepoints being created; you may simply delete them if they are of no use.

If your target agent supports a *circular trace buffer*, then you can run a trace experiment indefinitely without filling the trace buffer; when space runs out, the agent deletes already-collected trace frames, oldest first, until there is enough room to continue collecting. This is especially useful if your tracepoints are being hit too often, and your trace gets terminated prematurely because the buffer is full. To ask for a circular trace buffer, simply set '`circular-trace-buffer`' to on. You can set this at any time, including during tracing; if the agent can do it, it will change buffer handling on the fly, otherwise it will not take effect until the next run.

`set circular-trace-buffer on`
`set circular-trace-buffer off`
: Choose whether a tracing run should use a linear or circular buffer for trace data. A linear buffer will not lose any trace data, but may fill up prematurely, while a circular buffer will discard old trace data, but it will have always room for the latest tracepoint hits.

`show circular-trace-buffer`
: Show the current choice for the trace buffer. Note that this may not match the agent's current buffer handling, nor is it guaranteed to match the setting that might have been in effect during a past run, for instance if you are looking at frames from a trace file.

`set trace-buffer-size` *n*
`set trace-buffer-size unlimited`
: Request that the target use a trace buffer of *n* bytes. Not all targets will honor the request; they may have a compiled-in size for the trace buffer, or some other limitation. Set to a value of `unlimited` or `-1` to let the target use whatever size it likes. This is also the default.

`show trace-buffer-size`
: Show the current requested size for the trace buffer. Note that this will only match the actual size if the target supports size-setting, and was able to handle the requested size. For instance, if the target can only change buffer size between runs, this variable will not reflect the change until the next run starts. Use `tstatus` to get a report of the actual buffer size.

`set trace-user` *text*
`show trace-user`
`set trace-notes` *text*
: Set the trace run's notes.

`show trace-notes`
: Show the trace run's notes.

`set trace-stop-notes` *text*
: Set the trace run's stop notes. The handling of the note is as for `tstop` arguments; the set command is convenient way to fix a stop note that is mistaken or incomplete.

`show trace-stop-notes`
: Show the trace run's stop notes.

13.1.10 Tracepoint Restrictions

There are a number of restrictions on the use of tracepoints. As described above, tracepoint data gathering occurs on the target without interaction from GDB. Thus the full capabilities of the debugger are not available during data gathering, and then at data examination time, you will be limited by only having what was collected. The following items describe some common problems, but it is not exhaustive, and you may run into additional difficulties not mentioned here.

- Tracepoint expressions are intended to gather objects (lvalues). Thus the full flexibility of GDB's expression evaluator is not available. You cannot call functions, cast objects to aggregate types, access convenience variables or modify values (except by assignment to trace state variables). Some language features may implicitly call functions (for instance Objective-C fields with accessors), and therefore cannot be collected either.
- Collection of local variables, either individually or in bulk with `$locals` or `$args`, during `while-stepping` may behave erratically. The stepping action may enter a new scope (for instance by stepping into a function), or the location of the variable may change (for instance it is loaded into a register). The tracepoint data recorded uses the location information for the variables that is correct for the tracepoint location. When the tracepoint is created, it is not possible, in general, to determine where the steps of a `while-stepping` sequence will advance the program—particularly if a conditional branch is stepped.
- Collection of an incompletely-initialized or partially-destroyed object may result in something that GDB cannot display, or displays in a misleading way.
- When GDB displays a pointer to character it automatically dereferences the pointer to also display characters of the string being pointed to. However, collecting the pointer during tracing does not automatically collect the string. You need to explicitly dereference the pointer and provide size information if you want to collect not only the

pointer, but the memory pointed to. For example, `*ptr@50` can be used to collect the 50 element array pointed to by `ptr`.

- It is not possible to collect a complete stack backtrace at a tracepoint. Instead, you may collect the registers and a few hundred bytes from the stack pointer with something like `*(unsigned char *)$esp@300` (adjust to use the name of the actual stack pointer register on your target architecture, and the amount of stack you wish to capture). Then the `backtrace` command will show a partial backtrace when using a trace frame. The number of stack frames that can be examined depends on the sizes of the frames in the collected stack. Note that if you ask for a block so large that it goes past the bottom of the stack, the target agent may report an error trying to read from an invalid address.
- If you do not collect registers at a tracepoint, GDB can infer that the value of `$pc` must be the same as the address of the tracepoint and use that when you are looking at a trace frame for that tracepoint. However, this cannot work if the tracepoint has multiple locations (for instance if it was set in a function that was inlined), or if it has a `while-stepping` loop. In those cases GDB will warn you that it can't infer `$pc`, and default it to zero.

13.2 Using the Collected Data

After the tracepoint experiment ends, you use GDB commands for examining the trace data. The basic idea is that each tracepoint collects a trace *snapshot* every time it is hit and another snapshot every time it single-steps. All these snapshots are consecutively numbered from zero and go into a buffer, and you can examine them later. The way you examine them is to *focus* on a specific trace snapshot. When the remote stub is focused on a trace snapshot, it will respond to all GDB requests for memory and registers by reading from the buffer which belongs to that snapshot, rather than from *real* memory or registers of the program being debugged. This means that **all** GDB commands (`print`, `info registers`, `backtrace`, etc.) will behave as if we were currently debugging the program state as it was when the tracepoint occurred. Any requests for data that are not in the buffer will fail.

13.2.1 `tfind` *n*

The basic command for selecting a trace snapshot from the buffer is `tfind` *n*, which finds trace snapshot number *n*, counting from zero. If no argument *n* is given, the next snapshot is selected.

Here are the various forms of using the `tfind` command.

`tfind start`
: Find the first snapshot in the buffer. This is a synonym for `tfind 0` (since 0 is the number of the first snapshot).

`tfind none`
: Stop debugging trace snapshots, resume *live* debugging.

`tfind end` Same as '`tfind none`'.

`tfind` No argument means find the next trace snapshot or find the first one if no trace snapshot is selected.

`tfind -` Find the previous trace snapshot before the current one. This permits retracing earlier steps.

`tfind tracepoint` *num*
: Find the next snapshot associated with tracepoint *num*. Search proceeds forward from the last examined trace snapshot. If no argument *num* is given, it means find the next snapshot collected for the same tracepoint as the current snapshot.

`tfind pc` *addr*
: Find the next snapshot associated with the value *addr* of the program counter. Search proceeds forward from the last examined trace snapshot. If no argument *addr* is given, it means find the next snapshot with the same value of PC as the current snapshot.

`tfind outside` *addr1*`,` *addr2*
: Find the next snapshot whose PC is outside the given range of addresses (exclusive).

`tfind range` *addr1*`,` *addr2*
: Find the next snapshot whose PC is between *addr1* and *addr2* (inclusive).

`tfind line` [*file*`:`]*n*
: Find the next snapshot associated with the source line *n*. If the optional argument *file* is given, refer to line *n* in that source file. Search proceeds forward from the last examined trace snapshot. If no argument *n* is given, it means find the next line other than the one currently being examined; thus saying `tfind line` repeatedly can appear to have the same effect as stepping from line to line in a *live* debugging session.

The default arguments for the `tfind` commands are specifically designed to make it easy to scan through the trace buffer. For instance, `tfind` with no argument selects the next trace snapshot, and `tfind -` with no argument selects the previous trace snapshot. So, by giving one `tfind` command, and then simply hitting `RET` repeatedly you can examine all the trace snapshots in order. Or, by saying `tfind -` and then hitting `RET` repeatedly you can examine the snapshots in reverse order. The `tfind line` command with no argument selects the snapshot for the next source line executed. The `tfind pc` command with no argument selects the next snapshot with the same program counter (PC) as the current frame. The `tfind tracepoint` command with no argument selects the next trace snapshot collected by the same tracepoint as the current one.

In addition to letting you scan through the trace buffer manually, these commands make it easy to construct GDB scripts that scan through the trace buffer and print out whatever collected data you are interested in. Thus, if we want to examine the PC, FP, and SP registers from each trace frame in the buffer, we can say this:

```
(gdb) tfind start
(gdb) while ($trace_frame != -1)
> printf "Frame %d, PC = %08X, SP = %08X, FP = %08X\n", \
          $trace_frame, $pc, $sp, $fp
> tfind
> end
```

```
Frame 0, PC = 0020DC64, SP = 0030BF3C, FP = 0030BF44
Frame 1, PC = 0020DC6C, SP = 0030BF38, FP = 0030BF44
Frame 2, PC = 0020DC70, SP = 0030BF34, FP = 0030BF44
Frame 3, PC = 0020DC74, SP = 0030BF30, FP = 0030BF44
Frame 4, PC = 0020DC78, SP = 0030BF2C, FP = 0030BF44
Frame 5, PC = 0020DC7C, SP = 0030BF28, FP = 0030BF44
Frame 6, PC = 0020DC80, SP = 0030BF24, FP = 0030BF44
Frame 7, PC = 0020DC84, SP = 0030BF20, FP = 0030BF44
Frame 8, PC = 0020DC88, SP = 0030BF1C, FP = 0030BF44
Frame 9, PC = 0020DC8E, SP = 0030BF18, FP = 0030BF44
Frame 10, PC = 00203F6C, SP = 0030BE3C, FP = 0030BF14
```

Or, if we want to examine the variable `X` at each source line in the buffer:

```
(gdb) tfind start
(gdb) while ($trace_frame != -1)
> printf "Frame %d, X == %d\n", $trace_frame, X
> tfind line
> end

Frame 0, X = 1
Frame 7, X = 2
Frame 13, X = 255
```

13.2.2 `tdump`

This command takes no arguments. It prints all the data collected at the current trace snapshot.

```
(gdb) trace 444
(gdb) actions
Enter actions for tracepoint #2, one per line:
> collect $regs, $locals, $args, gdb_long_test
> end

(gdb) tstart

(gdb) tfind line 444
#0  gdb_test (p1=0x11, p2=0x22, p3=0x33, p4=0x44, p5=0x55, p6=0x66)
at gdb_test.c:444
444         printp( "%s: arguments = 0x%X 0x%X 0x%X 0x%X 0x%X 0x%X\n", )

(gdb) tdump
Data collected at tracepoint 2, trace frame 1:
d0              0xc4aa0085       -995491707
d1              0x18     24
d2              0x80     128
d3              0x33     51
d4              0x71aea3d        119204413
d5              0x22     34
d6              0xe0     224
d7              0x380035 3670069
a0              0x19e24a 1696330
a1              0x3000668        50333288
a2              0x100    256
a3              0x322000 3284992
a4              0x3000698        50333336
a5              0x1ad3cc 1758156
fp              0x30bf3c 0x30bf3c
sp              0x30bf34 0x30bf34
```

```
ps             0x0      0
pc             0x20b2c8 0x20b2c8
fpcontrol      0x0      0
fpstatus       0x0      0
fpiaddr        0x0      0
p = 0x20e5b4 "gdb-test"
p1 = (void *) 0x11
p2 = (void *) 0x22
p3 = (void *) 0x33
p4 = (void *) 0x44
p5 = (void *) 0x55
p6 = (void *) 0x66
gdb_long_test = 17 '\021'

(gdb)
```

`tdump` works by scanning the tracepoint's current collection actions and printing the value of each expression listed. So `tdump` can fail, if after a run, you change the tracepoint's actions to mention variables that were not collected during the run.

Also, for tracepoints with `while-stepping` loops, `tdump` uses the collected value of `$pc` to distinguish between trace frames that were collected at the tracepoint hit, and frames that were collected while stepping. This allows it to correctly choose whether to display the basic list of collections, or the collections from the body of the while-stepping loop. However, if `$pc` was not collected, then `tdump` will always attempt to dump using the basic collection list, and may fail if a while-stepping frame does not include all the same data that is collected at the tracepoint hit.

13.2.3 `save tracepoints` *`filename`*

This command saves all current tracepoint definitions together with their actions and passcounts, into a file *`filename`* suitable for use in a later debugging session. To read the saved tracepoint definitions, use the `source` command (see Section 23.1.3 [Command Files], page 371). The `save-tracepoints` command is a deprecated alias for `save tracepoints`

13.3 Convenience Variables for Tracepoints

`(int) $trace_frame`
: The current trace snapshot (a.k.a. *frame*) number, or -1 if no snapshot is selected.

`(int) $tracepoint`
: The tracepoint for the current trace snapshot.

`(int) $trace_line`
: The line number for the current trace snapshot.

`(char []) $trace_file`
: The source file for the current trace snapshot.

`(char []) $trace_func`
: The name of the function containing `$tracepoint`.

Note: `$trace_file` is not suitable for use in `printf`, use `output` instead.

Here's a simple example of using these convenience variables for stepping through all the trace snapshots and printing some of their data. Note that these are not the same as trace state variables, which are managed by the target.

```
(gdb) tfind start

(gdb) while $trace_frame != -1
> output $trace_file
> printf ", line %d (tracepoint #%d)\n", $trace_line, $tracepoint
> tfind
> end
```

13.4 Using Trace Files

In some situations, the target running a trace experiment may no longer be available; perhaps it crashed, or the hardware was needed for a different activity. To handle these cases, you can arrange to dump the trace data into a file, and later use that file as a source of trace data, via the `target tfile` command.

`tsave [ -r ]` *filename*
`tsave [-ctf]` *dirname*

Save the trace data to *filename*. By default, this command assumes that *filename* refers to the host filesystem, so if necessary GDB will copy raw trace data up from the target and then save it. If the target supports it, you can also supply the optional argument `-r` ("remote") to direct the target to save the data directly into *filename* in its own filesystem, which may be more efficient if the trace buffer is very large. (Note, however, that `target tfile` can only read from files accessible to the host.) By default, this command will save trace frame in tfile format. You can supply the optional argument `-ctf` to save data in CTF format. The *Common Trace Format* (CTF) is proposed as a trace format that can be shared by multiple debugging and tracing tools. Please go to '`http://www.efficios.com/ctf`' to get more information.

`target tfile` *filename*
`target ctf` *dirname*

Use the file named *filename* or directory named *dirname* as a source of trace data. Commands that examine data work as they do with a live target, but it is not possible to run any new trace experiments. `tstatus` will report the state of the trace run at the moment the data was saved, as well as the current trace frame you are examining. Both *filename* and *dirname* must be on a filesystem accessible to the host.

```
(gdb) target ctf ctf.ctf
(gdb) tfind
Found trace frame 0, tracepoint 2
39            ++a;  /* set tracepoint 1 here */
(gdb) tdump
Data collected at tracepoint 2, trace frame 0:
i = 0
a = 0
b = 1 '\001'
c = {"123", "456", "789", "123", "456", "789"}
d = {{{a = 1, b = 2}, {a = 3, b = 4}}, {{a = 5, b = 6}, {a = 7, b = 8}}}
(gdb) p b
```

```
$1 = 1
```

14 Debugging Programs That Use Overlays

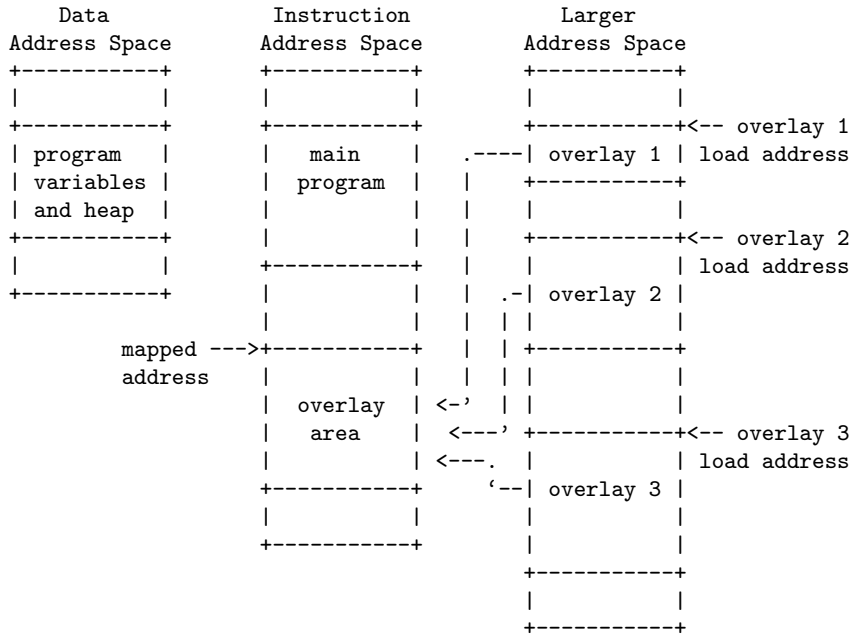
If your program is too large to fit completely in your target system's memory, you can sometimes use *overlays* to work around this problem. GDB provides some support for debugging programs that use overlays.

14.1 How Overlays Work

Suppose you have a computer whose instruction address space is only 64 kilobytes long, but which has much more memory which can be accessed by other means: special instructions, segment registers, or memory management hardware, for example. Suppose further that you want to adapt a program which is larger than 64 kilobytes to run on this system.

One solution is to identify modules of your program which are relatively independent, and need not call each other directly; call these modules *overlays*. Separate the overlays from the main program, and place their machine code in the larger memory. Place your main program in instruction memory, but leave at least enough space there to hold the largest overlay as well.

Now, to call a function located in an overlay, you must first copy that overlay's machine code from the large memory into the space set aside for it in the instruction memory, and then jump to its entry point there.



A code overlay

The diagram (see [A code overlay], page 211) shows a system with separate data and instruction address spaces. To map an overlay, the program copies its code from the larger address space to the instruction address space. Since the overlays shown here all use the same mapped address, only one may be mapped at a time. For a system with a single address space for data and instructions, the diagram would be similar, except that the program variables and heap would share an address space with the main program and the overlay area.

An overlay loaded into instruction memory and ready for use is called a *mapped* overlay; its *mapped address* is its address in the instruction memory. An overlay not present (or only partially present) in instruction memory is called *unmapped*; its *load address* is its address in the larger memory. The mapped address is also called the *virtual memory address*, or *VMA*; the load address is also called the *load memory address*, or *LMA*.

Unfortunately, overlays are not a completely transparent way to adapt a program to limited instruction memory. They introduce a new set of global constraints you must keep in mind as you design your program:

- Before calling or returning to a function in an overlay, your program must make sure that overlay is actually mapped. Otherwise, the call or return will transfer control to the right address, but in the wrong overlay, and your program will probably crash.
- If the process of mapping an overlay is expensive on your system, you will need to choose your overlays carefully to minimize their effect on your program's performance.
- The executable file you load onto your system must contain each overlay's instructions, appearing at the overlay's load address, not its mapped address. However, each overlay's instructions must be relocated and its symbols defined as if the overlay were at its mapped address. You can use GNU linker scripts to specify different load and relocation addresses for pieces of your program; see Section "Overlay Description" in *Using ld: the GNU linker*.
- The procedure for loading executable files onto your system must be able to load their contents into the larger address space as well as the instruction and data spaces.

The overlay system described above is rather simple, and could be improved in many ways:

- If your system has suitable bank switch registers or memory management hardware, you could use those facilities to make an overlay's load area contents simply appear at their mapped address in instruction space. This would probably be faster than copying the overlay to its mapped area in the usual way.
- If your overlays are small enough, you could set aside more than one overlay area, and have more than one overlay mapped at a time.
- You can use overlays to manage data, as well as instructions. In general, data overlays are even less transparent to your design than code overlays: whereas code overlays only require care when you call or return to functions, data overlays require care every time you access the data. Also, if you change the contents of a data overlay, you must copy its contents back out to its load address before you can copy a different data overlay into the same mapped area.

14.2 Overlay Commands

To use GDB's overlay support, each overlay in your program must correspond to a separate section of the executable file. The section's virtual memory address and load memory address must be the overlay's mapped and load addresses. Identifying overlays with sections allows GDB to determine the appropriate address of a function or variable, depending on whether the overlay is mapped or not.

GDB's overlay commands all start with the word `overlay`; you can abbreviate this as `ov` or `ovly`. The commands are:

`overlay off`
: Disable GDB's overlay support. When overlay support is disabled, GDB assumes that all functions and variables are always present at their mapped addresses. By default, GDB's overlay support is disabled.

`overlay manual`
: Enable *manual* overlay debugging. In this mode, GDB relies on you to tell it which overlays are mapped, and which are not, using the `overlay map-overlay` and `overlay unmap-overlay` commands described below.

`overlay map-overlay` *overlay*
`overlay map` *overlay*
: Tell GDB that *overlay* is now mapped; *overlay* must be the name of the object file section containing the overlay. When an overlay is mapped, GDB assumes it can find the overlay's functions and variables at their mapped addresses. GDB assumes that any other overlays whose mapped ranges overlap that of *overlay* are now unmapped.

`overlay unmap-overlay` *overlay*
`overlay unmap` *overlay*
: Tell GDB that *overlay* is no longer mapped; *overlay* must be the name of the object file section containing the overlay. When an overlay is unmapped, GDB assumes it can find the overlay's functions and variables at their load addresses.

`overlay auto`
: Enable *automatic* overlay debugging. In this mode, GDB consults a data structure the overlay manager maintains in the inferior to see which overlays are mapped. For details, see Section 14.3 [Automatic Overlay Debugging], page 214.

`overlay load-target`
`overlay load`
: Re-read the overlay table from the inferior. Normally, GDB re-reads the table GDB automatically each time the inferior stops, so this command should only be necessary if you have changed the overlay mapping yourself using GDB. This command is only useful when using automatic overlay debugging.

`overlay list-overlays`
`overlay list`
: Display a list of the overlays currently mapped, along with their mapped addresses, load addresses, and sizes.

Normally, when GDB prints a code address, it includes the name of the function the address falls in:

```
(gdb) print main
$3 = {int ()} 0x11a0 <main>
```

When overlay debugging is enabled, GDB recognizes code in unmapped overlays, and prints the names of unmapped functions with asterisks around them. For example, if `foo` is a function in an unmapped overlay, GDB prints it this way:

```
(gdb) overlay list
No sections are mapped.
(gdb) print foo
```

```
$5 = {int (int)} 0x100000 <*foo*>
```

When `foo`'s overlay is mapped, GDB prints the function's name normally:

```
(gdb) overlay list
Section .ov.foo.text, loaded at 0x100000 - 0x100034,
        mapped at 0x1016 - 0x104a
(gdb) print foo
$6 = {int (int)} 0x1016 <foo>
```

When overlay debugging is enabled, GDB can find the correct address for functions and variables in an overlay, whether or not the overlay is mapped. This allows most GDB commands, like `break` and `disassemble`, to work normally, even on unmapped code. However, GDB's breakpoint support has some limitations:

- You can set breakpoints in functions in unmapped overlays, as long as GDB can write to the overlay at its load address.
- GDB can not set hardware or simulator-based breakpoints in unmapped overlays. However, if you set a breakpoint at the end of your overlay manager (and tell GDB which overlays are now mapped, if you are using manual overlay management), GDB will re-set its breakpoints properly.

14.3 Automatic Overlay Debugging

GDB can automatically track which overlays are mapped and which are not, given some simple co-operation from the overlay manager in the inferior. If you enable automatic overlay debugging with the `overlay auto` command (see Section 14.2 [Overlay Commands], page 212), GDB looks in the inferior's memory for certain variables describing the current state of the overlays.

Here are the variables your overlay manager must define to support GDB's automatic overlay debugging:

`_ovly_table`:
: This variable must be an array of the following structures:

```
struct
{
  /* The overlay's mapped address.  */
  unsigned long vma;

  /* The size of the overlay, in bytes.  */
  unsigned long size;

  /* The overlay's load address.  */
  unsigned long lma;

  /* Non-zero if the overlay is currently mapped;
     zero otherwise.  */
  unsigned long mapped;
}
```

`_novlys`:
: This variable must be a four-byte signed integer, holding the total number of elements in `_ovly_table`.

To decide whether a particular overlay is mapped or not, GDB looks for an entry in `_ovly_table` whose `vma` and `lma` members equal the VMA and LMA of the overlay's section

in the executable file. When GDB finds a matching entry, it consults the entry's `mapped` member to determine whether the overlay is currently mapped.

In addition, your overlay manager may define a function called `_ovly_debug_event`. If this function is defined, GDB will silently set a breakpoint there. If the overlay manager then calls this function whenever it has changed the overlay table, this will enable GDB to accurately keep track of which overlays are in program memory, and update any breakpoints that may be set in overlays. This will allow breakpoints to work even if the overlays are kept in ROM or other non-writable memory while they are not being executed.

14.4 Overlay Sample Program

When linking a program which uses overlays, you must place the overlays at their load addresses, while relocating them to run at their mapped addresses. To do this, you must write a linker script (see Section "Overlay Description" in *Using ld: the GNU linker*). Unfortunately, since linker scripts are specific to a particular host system, target architecture, and target memory layout, this manual cannot provide portable sample code demonstrating GDB's overlay support.

However, the GDB source distribution does contain an overlaid program, with linker scripts for a few systems, as part of its test suite. The program consists of the following files from `gdb/testsuite/gdb.base`:

`overlays.c`
: The main program file.

`ovlymgr.c`
: A simple overlay manager, used by `overlays.c`.

`foo.c`
`bar.c`
`baz.c`
`grbx.c`
: Overlay modules, loaded and used by `overlays.c`.

`d10v.ld`
`m32r.ld`
: Linker scripts for linking the test program on the `d10v-elf` and `m32r-elf` targets.

You can build the test program using the `d10v-elf` GCC cross-compiler like this:

```
$ d10v-elf-gcc -g -c overlays.c
$ d10v-elf-gcc -g -c ovlymgr.c
$ d10v-elf-gcc -g -c foo.c
$ d10v-elf-gcc -g -c bar.c
$ d10v-elf-gcc -g -c baz.c
$ d10v-elf-gcc -g -c grbx.c
$ d10v-elf-gcc -g overlays.o ovlymgr.o foo.o bar.o \
                  baz.o grbx.o -Wl,-Td10v.ld -o overlays
```

The build process is identical for any other architecture, except that you must substitute the appropriate compiler and linker script for the target system for `d10v-elf-gcc` and `d10v.ld`.

15 Using GDB with Different Languages

Although programming languages generally have common aspects, they are rarely expressed in the same manner. For instance, in ANSI C, dereferencing a pointer `p` is accomplished by `*p`, but in Modula-2, it is accomplished by `p^`. Values can also be represented (and displayed) differently. Hex numbers in C appear as '`0x1ae`', while in Modula-2 they appear as '`1AEH`'.

Language-specific information is built into GDB for some languages, allowing you to express operations like the above in your program's native language, and allowing GDB to output values in a manner consistent with the syntax of your program's native language. The language you use to build expressions is called the *working language*.

15.1 Switching Between Source Languages

There are two ways to control the working language—either have GDB set it automatically, or select it manually yourself. You can use the `set language` command for either purpose. On startup, GDB defaults to setting the language automatically. The working language is used to determine how expressions you type are interpreted, how values are printed, etc.

In addition to the working language, every source file that GDB knows about has its own working language. For some object file formats, the compiler might indicate which language a particular source file is in. However, most of the time GDB infers the language from the name of the file. The language of a source file controls whether C++ names are demangled—this way `backtrace` can show each frame appropriately for its own language. There is no way to set the language of a source file from within GDB, but you can set the language associated with a filename extension. See Section 15.2 [Displaying the Language], page 218.

This is most commonly a problem when you use a program, such as `cfront` or `f2c`, that generates C but is written in another language. In that case, make the program use `#line` directives in its C output; that way GDB will know the correct language of the source code of the original program, and will display that source code, not the generated C code.

15.1.1 List of Filename Extensions and Languages

If a source file name ends in one of the following extensions, then GDB infers that its language is the one indicated.

`.ada`
`.ads`
`.adb`
`.a` Ada source file.

`.c` C source file

`.C`
`.cc`
`.cp`
`.cpp`
`.cxx`
`.c++` C++ source file

`.d` D source file

`.m` Objective-C source file

`.f`
`.F` Fortran source file

`.mod` Modula-2 source file

`.s`
`.S` Assembler source file. This actually behaves almost like C, but GDB does not skip over function prologues when stepping.

In addition, you may set the language associated with a filename extension. See Section 15.2 [Displaying the Language], page 218.

15.1.2 Setting the Working Language

If you allow GDB to set the language automatically, expressions are interpreted the same way in your debugging session and your program.

If you wish, you may set the language manually. To do this, issue the command '`set language` *lang*', where *lang* is the name of a language, such as `c` or `modula-2`. For a list of the supported languages, type '`set language`'.

Setting the language manually prevents GDB from updating the working language automatically. This can lead to confusion if you try to debug a program when the working language is not the same as the source language, when an expression is acceptable to both languages—but means different things. For instance, if the current source file were written in C, and GDB was parsing Modula-2, a command such as:

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

might not have the effect you intended. In C, this means to add `b` and `c` and place the result in `a`. The result printed would be the value of `a`. In Modula-2, this means to compare `a` to the result of `b+c`, yielding a `BOOLEAN` value.

15.1.3 Having GDB Infer the Source Language

To have GDB set the working language automatically, use '`set language local`' or '`set language auto`'. GDB then infers the working language. That is, when your program stops in a frame (usually by encountering a breakpoint), GDB sets the working language to the language recorded for the function in that frame. If the language for a frame is unknown (that is, if the function or block corresponding to the frame was defined in a source file that does not have a recognized extension), the current working language is not changed, and GDB issues a warning.

This may not seem necessary for most programs, which are written entirely in one source language. However, program modules and libraries written in one source language can be used by a main program written in a different source language. Using '`set language auto`' in this case frees you from having to set the working language manually.

15.2 Displaying the Language

The following commands help you find out which language is the working language, and also what language source files were written in.

`show language`
: Display the current working language. This is the language you can use with commands such as `print` to build and compute expressions that may involve variables in your program.

`info frame`
: Display the source language for this frame. This language becomes the working language if you use an identifier from this frame. See Section 8.4 [Information about a Frame], page 113, to identify the other information listed here.

`info source`
: Display the source language of this source file. See Chapter 16 [Examining the Symbol Table], page 251, to identify the other information listed here.

In unusual circumstances, you may have source files with extensions not in the standard list. You can then set the extension associated with a language explicitly:

`set extension-language` *`ext language`*
: Tell GDB that source files with extension *ext* are to be assumed as written in the source language *language*.

`info extensions`
: List all the filename extensions and the associated languages.

15.3 Type and Range Checking

Some languages are designed to guard you against making seemingly common errors through a series of compile- and run-time checks. These include checking the type of arguments to functions and operators and making sure mathematical overflows are caught at run time. Checks such as these help to ensure a program's correctness once it has been compiled by eliminating type mismatches and providing active checks for range errors when your program is running.

By default GDB checks for these errors according to the rules of the current source language. Although GDB does not check the statements in your program, it can check expressions entered directly into GDB for evaluation via the `print` command, for example.

15.3.1 An Overview of Type Checking

Some languages, such as C and C++, are strongly typed, meaning that the arguments to operators and functions have to be of the correct type, otherwise an error occurs. These checks prevent type mismatch errors from ever causing any run-time problems. For example,

```
int klass::my_method(char *b) { return  b ? 1 : 2; }

(gdb) print obj.my_method (0)
$1 = 2
```

but

```
(gdb) print obj.my_method (0x1234)
Cannot resolve method klass::my_method to any overloaded instance
```

The second example fails because in C++ the integer constant '`0x1234`' is not type-compatible with the pointer parameter type.

For the expressions you use in GDB commands, you can tell GDB to not enforce strict type checking or to treat any mismatches as errors and abandon the expression; When type checking is disabled, GDB successfully evaluates expressions like the second example above.

Even if type checking is off, there may be other reasons related to type that prevent GDB from evaluating an expression. For instance, GDB does not know how to add an `int` and a `struct foo`. These particular type errors have nothing to do with the language in use and usually arise from expressions which make little sense to evaluate anyway.

GDB provides some additional commands for controlling type checking:

`set check type on`
`set check type off`
: Set strict type checking on or off. If any type mismatches occur in evaluating an expression while type checking is on, GDB prints a message and aborts evaluation of the expression.

`show check type`
: Show the current setting of type checking and whether GDB is enforcing strict type checking rules.

15.3.2 An Overview of Range Checking

In some languages (such as Modula-2), it is an error to exceed the bounds of a type; this is enforced with run-time checks. Such range checking is meant to ensure program correctness by making sure computations do not overflow, or indices on an array element access do not exceed the bounds of the array.

For expressions you use in GDB commands, you can tell GDB to treat range errors in one of three ways: ignore them, always treat them as errors and abandon the expression, or issue warnings but evaluate the expression anyway.

A range error can result from numerical overflow, from exceeding an array index bound, or when you type a constant that is not a member of any type. Some languages, however, do not treat overflows as an error. In many implementations of C, mathematical overflow causes the result to "wrap around" to lower values—for example, if m is the largest integer value, and s is the smallest, then

$$m + 1 \Rightarrow s$$

This, too, is specific to individual languages, and in some cases specific to individual compilers or machines. See Section 15.4 [Supported Languages], page 221, for further details on specific languages.

GDB provides some additional commands for controlling the range checker:

`set check range auto`
: Set range checking on or off based on the current working language. See Section 15.4 [Supported Languages], page 221, for the default settings for each language.

`set check range on`
`set check range off`
: Set range checking on or off, overriding the default setting for the current working language. A warning is issued if the setting does not match the language

default. If a range error occurs and range checking is on, then a message is printed and evaluation of the expression is aborted.

`set check range warn`
: Output messages when the GDB range checker detects a range error, but attempt to evaluate the expression anyway. Evaluating the expression may still be impossible for other reasons, such as accessing memory that the process does not own (a typical example from many Unix systems).

`show check range`
: Show the current setting of the range checker, and whether or not it is being set automatically by GDB.

15.4 Supported Languages

GDB supports C, C++, D, Go, Objective-C, Fortran, OpenCL C, Pascal, Rust, assembly, Modula-2, and Ada. Some GDB features may be used in expressions regardless of the language you use: the GDB `@` and `::` operators, and the '`{type}addr`' construct (see Section 10.1 [Expressions], page 137) can be used with the constructs of any supported language.

The following sections detail to what degree each source language is supported by GDB. These sections are not meant to be language tutorials or references, but serve only as a reference guide to what the GDB expression parser accepts, and what input and output formats should look like for different languages. There are many good books written on each of these languages; please look to these for a language reference or tutorial.

15.4.1 C and C++

Since C and C++ are so closely related, many features of GDB apply to both languages. Whenever this is the case, we discuss those languages together.

The C++ debugging facilities are jointly implemented by the C++ compiler and GDB. Therefore, to debug your C++ code effectively, you must compile your C++ programs with a supported C++ compiler, such as GNU `g++`, or the HP ANSI C++ compiler (`aCC`).

15.4.1.1 C and C++ Operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on structures. Operators are often defined on groups of types.

For the purposes of C and C++, the following definitions hold:

- *Integral types* include `int` with any of its storage-class specifiers; `char`; `enum`; and, for C++, `bool`.
- *Floating-point types* include `float`, `double`, and `long double` (if supported by the target platform).
- *Pointer types* include all types defined as `(type *)`.
- *Scalar types* include all of the above.

The following operators are supported. They are listed here in order of increasing precedence:

`,`
: The comma or sequencing operator. Expressions in a comma-separated list are evaluated from left to right, with the result of the entire expression being the last expression evaluated.

`=` Assignment. The value of an assignment expression is the value assigned. Defined on scalar types.

`op=` Used in an expression of the form `a op= b`, and translated to `a = a op b`. `op=` and `=` have the same precedence. The operator *op* is any one of the operators `|`, `^`, `&`, `<<`, `>>`, `+`, `-`, `*`, `/`, `%`.

`?:` The ternary operator. `a ? b : c` can be thought of as: if *a* then *b* else *c*. The argument *a* should be of an integral type.

`||` Logical OR. Defined on integral types.

`&&` Logical AND. Defined on integral types.

`|` Bitwise OR. Defined on integral types.

`^` Bitwise exclusive-OR. Defined on integral types.

`&` Bitwise AND. Defined on integral types.

`==`, `!=` Equality and inequality. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<`, `>`, `<=`, `>=`
Less than, greater than, less than or equal, greater than or equal. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<<`, `>>` left shift, and right shift. Defined on integral types.

`@` The GDB "artificial array" operator (see Section 10.1 [Expressions], page 137).

`+`, `-` Addition and subtraction. Defined on integral types, floating-point types and pointer types.

`*`, `/`, `%` Multiplication, division, and modulus. Multiplication and division are defined on integral and floating-point types. Modulus is defined on integral types.

`++`, `--` Increment and decrement. When appearing before a variable, the operation is performed before the variable is used in an expression; when appearing after it, the variable's value is used before the operation takes place.

`*` Pointer dereferencing. Defined on pointer types. Same precedence as `++`.

`&` Address operator. Defined on variables. Same precedence as `++`.

For debugging C++, GDB implements a use of '`&`' beyond what is allowed in the C++ language itself: you can use '`&(&ref)`' to examine the address where a C++ reference variable (declared with '`&ref`') is stored.

`-` Negative. Defined on integral and floating-point types. Same precedence as `++`.

`!` Logical negation. Defined on integral types. Same precedence as `++`.

`~` Bitwise complement operator. Defined on integral types. Same precedence as `++`.

`.`, `->` Structure member, and pointer-to-structure member. For convenience, GDB regards the two as equivalent, choosing whether to dereference a pointer based on the stored type information. Defined on `struct` and `union` data.

`.*`, `->*` Dereferences of pointers to members.

`[]` Array indexing. `a[i]` is defined as `*(a+i)`. Same precedence as `->`.

`()` Function parameter list. Same precedence as `->`.

`::` C++ scope resolution operator. Defined on `struct`, `union`, and `class` types.

`::` Doubled colons also represent the GDB scope operator (see Section 10.1 [Expressions], page 137). Same precedence as `::`, above.

If an operator is redefined in the user code, GDB usually attempts to invoke the redefined version instead of using the operator's predefined meaning.

15.4.1.2 C and C++ Constants

GDB allows you to express the constants of C and C++ in the following ways:

- Integer constants are a sequence of digits. Octal constants are specified by a leading '`0`' (i.e. zero), and hexadecimal constants by a leading '`0x`' or '`0X`'. Constants may also end with a letter '`l`', specifying that the constant should be treated as a `long` value.
- Floating point constants are a sequence of digits, followed by a decimal point, followed by a sequence of digits, and optionally followed by an exponent. An exponent is of the form: '`e`[`[+]|-`]*nnn*', where *nnn* is another sequence of digits. The '`+`' is optional for positive exponents. A floating-point constant may also end with a letter '`f`' or '`F`', specifying that the constant should be treated as being of the `float` (as opposed to the default `double`) type; or with a letter '`l`' or '`L`', which specifies a `long double` constant.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes (`'`), or a number—the ordinal value of the corresponding character (usually its ASCII value). Within quotes, the single character may be represented by a letter or by *escape sequences*, which are of the form '`\`*nnn*', where *nnn* is the octal representation of the character's ordinal value; or of the form '`\`*x*', where '*x*' is a predefined special character—for example, '`\n`' for newline.

  Wide character constants can be written by prefixing a character constant with '`L`', as in C. For example, '`L'x'`' is the wide form of '`x`'. The target wide character set is used when computing the value of this constant (see Section 10.21 [Character Sets], page 177).
- String constants are a sequence of character constants surrounded by double quotes (`"`). Any valid character constant (as described above) may appear. Double quotes within the string must be preceded by a backslash, so for instance '`"a\"b'c"`' is a string of five characters.

  Wide string constants can be written by prefixing a string constant with '`L`', as in C. The target wide character set is used when computing the value of this constant (see Section 10.21 [Character Sets], page 177).
- Pointer constants are an integral value. You can also write pointers to constants using the C operator '`&`'.
- Array constants are comma-separated lists surrounded by braces '`{`' and '`}`'; for example, '`{1,2,3}`' is a three-element array of integers, '`{{1,2}, {3,4}, {5,6}}`' is a

three-by-two array, and '`{&"hi", &"there", &"fred"}`' is a three-element array of pointers.

15.4.1.3 C++ Expressions

GDB expression handling can interpret most C++ expressions.

> *Warning:* GDB can only debug C++ code if you use the proper compiler and the proper debug format. Currently, GDB works best when debugging C++ code that is compiled with the most recent version of GCC possible. The DWARF debugging format is preferred; GCC defaults to this on most popular platforms. Other compilers and/or debug formats are likely to work badly or not at all when using GDB to debug C++ code. See Section 4.1 [Compilation], page 31.

1. Member function calls are allowed; you can use expressions like

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

2. While a member function is active (in the selected stack frame), your expressions have the same namespace available as the member function; that is, GDB allows implicit references to the class instance pointer `this` following the same rules as C++. `using` declarations in the current scope are also respected by GDB.

3. You can call overloaded functions; GDB resolves the function call to the right definition, with some restrictions. GDB does not perform overload resolution involving user-defined type conversions, calls to constructors, or instantiations of templates that do not exist in the program. It also cannot handle ellipsis argument lists or default arguments.

   It does perform integral conversions and promotions, floating-point promotions, arithmetic conversions, pointer conversions, conversions of class objects to base classes, and standard conversions such as those of functions or arrays to pointers; it requires an exact match on the number of function arguments.

   Overload resolution is always performed, unless you have specified `set overload-resolution off`. See Section 15.4.1.7 [GDB Features for C++], page 225.

   You must specify `set overload-resolution off` in order to use an explicit function signature to call an overloaded function, as in

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

   The GDB command-completion facility can simplify this; see Section 3.3 [Command Completion], page 24.

4. GDB understands variables declared as C++ lvalue or rvalue references; you can use them in expressions just as you do in C++ source—they are automatically dereferenced.

   In the parameter list shown when GDB displays a frame, the values of reference variables are not displayed (unlike other variables); this avoids clutter, since references are often used for large structures. The *address* of a reference variable is always shown, unless you have specified '`set print address off`'.

5. GDB supports the C++ name resolution operator `::`—your expressions can use it just as expressions in your program do. Since one scope may be defined in another, you can use `::` repeatedly if necessary, for example in an expression like '*`scope1`*`::`*`scope2`*`::`*`name`*'. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see Section 10.3 [Program Variables], page 138).

6. GDB performs argument-dependent lookup, following the C++ specification.

15.4.1.4 C and C++ Defaults

If you allow GDB to set range checking automatically, it defaults to `off` whenever the working language changes to C or C++. This happens regardless of whether you or GDB selects the working language.

If you allow GDB to set the language automatically, it recognizes source files whose names end with `.c`, `.C`, or `.cc`, etc, and when GDB enters code compiled from one of these files, it sets the working language to C or C++. See Section 15.1.3 [Having GDB Infer the Source Language], page 218, for further details.

15.4.1.5 C and C++ Type and Range Checks

By default, when GDB parses C or C++ expressions, strict type checking is used. However, if you turn type checking off, GDB will allow certain non-standard conversions, such as promoting integer constants to pointers.

Range checking, if turned on, is done on mathematical operations. Array indices are not checked, since they are often used to index a pointer that is not itself an array.

15.4.1.6 GDB and C

The `set print union` and `show print union` commands apply to the `union` type. When set to '`on`', any `union` that is inside a `struct` or `class` is also printed. Otherwise, it appears as '`{...}`'.

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function. See Section 10.1 [Expressions], page 137.

15.4.1.7 GDB Features for C++

Some GDB commands are particularly useful with C++, and some are designed specifically for use with C++. Here is a summary:

breakpoint menus
: When you want a breakpoint in a function whose name is overloaded, GDB has the capability to display a menu of possible breakpoint locations to help you specify which function definition you want. See Section 10.2 [Ambiguous Expressions], page 137.

`rbreak` *regex*
: Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not members of any special classes. See Section 5.1.1 [Setting Breakpoints], page 56.

`catch throw`
`catch rethrow`
`catch catch`
: Debug C++ exception handling using these commands. See Section 5.1.3 [Setting Catchpoints], page 66.

`ptype` *typename*
: Print inheritance relationships as well as other information for type *typename*. See Chapter 16 [Examining the Symbol Table], page 251.

`info vtbl` *expression*.
: The `info vtbl` command can be used to display the virtual method tables of the object computed by *expression*. This shows one entry per virtual table; there may be multiple virtual tables when multiple inheritance is in use.

`demangle` *name*
: Demangle *name*. See Chapter 16 [Symbols], page 251, for a more complete description of the `demangle` command.

`set print demangle`
`show print demangle`
`set print asm-demangle`
`show print asm-demangle`
: Control whether C++ symbols display in their source form, both when displaying code as C++ source and when displaying disassemblies. See Section 10.9 [Print Settings], page 149.

`set print object`
`show print object`
: Choose whether to print derived (actual) or declared types of objects. See Section 10.9 [Print Settings], page 149.

`set print vtbl`
`show print vtbl`
: Control the format for printing virtual function tables. See Section 10.9 [Print Settings], page 149. (The `vtbl` commands do not work on programs compiled with the HP ANSI C++ compiler (`aCC`).)

`set overload-resolution on`
: Enable overload resolution for C++ expression evaluation. The default is on. For overloaded functions, GDB evaluates the arguments and searches for a function whose signature matches the argument types, using the standard C++ conversion rules (see Section 15.4.1.3 [C++ Expressions], page 224, for details). If it cannot find a match, it emits a message.

`set overload-resolution off`
: Disable overload resolution for C++ expression evaluation. For overloaded functions that are not class member functions, GDB chooses the first function of the specified name that it finds in the symbol table, whether or not its arguments are of the correct type. For overloaded functions that are class member functions, GDB searches for a function whose signature *exactly* matches the argument types.

`show overload-resolution`
: Show the current setting of overload resolution.

Overloaded symbol names
: You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type *`symbol(types)`* rather than just *symbol*. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See Section 3.3 [Command Completion], page 24, for details on how to do this.

Breakpoints in template functions

Similar to how overloaded symbols are handled, GDB will ignore template parameter lists when it encounters a symbol which includes a C++ template. This permits setting breakpoints on families of template functions or functions whose parameters include template types.

The *-qualified* flag may be used to override this behavior, causing GDB to search for a specific function or type.

The GDB command-line word completion facility also understands template parameters and may be used to list available choices or finish template parameter lists for you. See Section 3.3 [Command Completion], page 24, for details on how to do this.

Breakpoints in functions with ABI tags

The GNU C++ compiler introduced the notion of ABI "tags", which correspond to changes in the ABI of a type, function, or variable that would not otherwise be reflected in a mangled name. See `https://developers.redhat.com/blog/2015/02/05/gcc5-and-the-c11-abi/` for more detail.

The ABI tags are visible in C++ demangled names. For example, a function that returns a std::string:

```
std::string function(int);
```

when compiled for the C++11 ABI is marked with the `cxx11` ABI tag, and GDB displays the symbol like this:

```
function[abi:cxx11](int)
```

You can set a breakpoint on such functions simply as if they had no tag. For example:

```
(gdb) b function(int)
Breakpoint 2 at 0x40060d: file main.cc, line 10.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0040060d in function[abi:cxx11](int)
                                           at main.cc:10
```

On the rare occasion you need to disambiguate between different ABI tags, you can do so by simply including the ABI tag in the function name, like:

```
(gdb) b ambiguous[abi:other_tag](int)
```

15.4.1.8 Decimal Floating Point format

GDB can examine, set and perform computations with numbers in decimal floating point format, which in the C language correspond to the `_Decimal32`, `_Decimal64` and `_Decimal128` types as specified by the extension to support decimal floating-point arithmetic.

There are two encodings in use, depending on the architecture: BID (Binary Integer Decimal) for x86 and x86-64, and DPD (Densely Packed Decimal) for PowerPC and S/390. GDB will use the appropriate encoding for the configured target.

Because of a limitation in `libdecnumber`, the library used by GDB to manipulate decimal floating point numbers, it is not possible to convert (using a cast, for example) integers wider than 32-bit to decimal float.

In addition, in order to imitate GDB's behaviour with binary floating point computations, error checking in decimal float operations ignores underflow, overflow and divide by zero exceptions.

In the PowerPC architecture, GDB provides a set of pseudo-registers to inspect `_Decimal128` values stored in floating point registers. See Section 21.4.6 [PowerPC], page 342, for more details.

15.4.2 D

GDB can be used to debug programs written in D and compiled with GDC, LDC or DMD compilers. Currently GDB supports only one D specific feature — dynamic arrays.

15.4.3 Go

GDB can be used to debug programs written in Go and compiled with `gccgo` or `6g` compilers.

Here is a summary of the Go-specific features and restrictions:

`The current Go package`
: The name of the current package does not need to be specified when specifying global variables and functions.

  For example, given the program:

  ```
  package main
  var myglob = "Shall we?"
  func main () {
    // ...
  }
  ```

  When stopped inside `main` either of these work:

  ```
  (gdb) p myglob
  (gdb) p main.myglob
  ```

`Builtin Go types`
: The `string` type is recognized by GDB and is printed as a string.

`Builtin Go functions`
: The GDB expression parser recognizes the `unsafe.Sizeof` function and handles it internally.

`Restrictions on Go expressions`
: All Go operators are supported except `&^`. The Go `_` "blank identifier" is not supported. Automatic dereferencing of pointers is not supported.

15.4.4 Objective-C

This section provides information about some commands and command options that are useful for debugging Objective-C code. See also Chapter 16 [Symbols], page 251, and Chapter 16 [Symbols], page 251, for a few more commands specific to Objective-C support.

15.4.4.1 Method Names in Commands

The following commands have been extended to accept Objective-C method names as line specifications:

- `clear`

- `break`
- `info line`
- `jump`
- `list`

A fully qualified Objective-C method name is specified as

```
-[Class methodName]
```

where the minus sign is used to indicate an instance method and a plus sign (not shown) is used to indicate a class method. The class name *Class* and method name *methodName* are enclosed in brackets, similar to the way messages are specified in Objective-C source code. For example, to set a breakpoint at the `create` instance method of class `Fruit` in the program currently being debugged, enter:

```
break -[Fruit create]
```

To list ten program lines around the `initialize` class method, enter:

```
list +[NSText initialize]
```

In the current version of GDB, the plus or minus sign is required. In future versions of GDB, the plus or minus sign will be optional, but you can use it to narrow the search. It is also possible to specify just a method name:

```
break create
```

You must specify the complete method name, including any colons. If your program's source files contain more than one `create` method, you'll be presented with a numbered list of classes that implement that method. Indicate your choice by number, or type '`0`' to exit if none apply.

As another example, to clear a breakpoint established at the `makeKeyAndOrderFront:` method of the `NSWindow` class, enter:

```
clear -[NSWindow makeKeyAndOrderFront:]
```

15.4.4.2 The Print Command With Objective-C

The print command has also been extended to accept methods. For example:

```
print -[object hash]
```

will tell GDB to send the `hash` message to *object* and print the result. Also, an additional command has been added, `print-object` or `po` for short, which is meant to print the description of an object. However, this command may only work with certain Objective-C libraries that have a particular hook function, `_NSPrintForDebugger`, defined.

15.4.5 OpenCL C

This section provides information about GDBs OpenCL C support.

15.4.5.1 OpenCL C Datatypes

GDB supports the builtin scalar and vector datatypes specified by OpenCL 1.1. In addition the half- and double-precision floating point data types of the `cl_khr_fp16` and `cl_khr_fp64` OpenCL extensions are also known to GDB.

15.4.5.2 OpenCL C Expressions

GDB supports accesses to vector components including the access as lvalue where possible. Since OpenCL C is based on C99 most C expressions supported by GDB can be used as well.

15.4.5.3 OpenCL C Operators

GDB supports the operators specified by OpenCL 1.1 for scalar and vector data types.

15.4.6 Fortran

GDB can be used to debug programs written in Fortran, but it currently supports only the features of Fortran 77 language.

Some Fortran compilers (GNU Fortran 77 and Fortran 95 compilers among them) append an underscore to the names of variables and functions. When you debug programs compiled by those compilers, you will need to refer to variables and functions with a trailing underscore.

15.4.6.1 Fortran Operators and Expressions

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on characters or other non- arithmetic types. Operators are often defined on groups of types.

`**` The exponentiation operator. It raises the first operand to the power of the second one.

`:` The range operator. Normally used in the form of array(low:high) to represent a section of array.

`%` The access component operator. Normally used to access elements in derived types. Also suitable for unions. As unions aren't part of regular Fortran, this can only happen when accessing a register that uses a gdbarch-defined union type.

`::` The scope operator. Normally used to access variables in modules or to set breakpoints on subroutines nested in modules or in other subroutines (internal subroutines).

15.4.6.2 Fortran Defaults

Fortran symbols are usually case-insensitive, so GDB by default uses case-insensitive matches for Fortran symbols. You can change that with the '`set case-insensitive`' command, see Chapter 16 [Symbols], page 251, for the details.

15.4.6.3 Special Fortran Commands

GDB has some commands to support Fortran-specific features, such as displaying common blocks.

`info common` [*common-name*]

This command prints the values contained in the Fortran `COMMON` block whose name is *common-name*. With no argument, the names of all `COMMON` blocks visible at the current program location are printed.

`set fortran repack-array-slices [on|off]`
`show fortran repack-array-slices`

When taking a slice from an array, a Fortran compiler can choose to either produce an array descriptor that describes the slice in place, or it may repack the slice, copying the elements of the slice into a new region of memory.

When this setting is on, then GDB will also repack array slices in some situations. When this setting is off, then GDB will create array descriptors for slices that reference the original data in place.

GDB will never repack an array slice if the data for the slice is contiguous within the original array.

GDB will always repack string slices if the data for the slice is non-contiguous within the original string as GDB does not support printing non-contiguous strings.

The default for this setting is `off`.

15.4.7 Pascal

Debugging Pascal programs which use sets, subranges, file variables, or nested functions does not currently work. GDB does not support entering expressions, printing values, or similar features using Pascal syntax.

The Pascal-specific command `set print pascal_static-members` controls whether static members of Pascal objects are displayed. See Section 10.9 [Print Settings], page 149.

15.4.8 Rust

GDB supports the Rust Programming Language (`https://www.rust-lang.org/`). Type- and value-printing, and expression parsing, are reasonably complete. However, there are a few peculiarities and holes to be aware of.

- Linespecs (see Section 9.2 [Specify Location], page 120) are never relative to the current crate. Instead, they act as if there were a global namespace of crates, somewhat similar to the way `extern crate` behaves.

  That is, if GDB is stopped at a breakpoint in a function in crate '`A`', module '`B`', then `break B::f` will attempt to set a breakpoint in a function named '`f`' in a crate named '`B`'.

  As a consequence of this approach, linespecs also cannot refer to items using '`self::`' or '`super::`'.

- Because GDB implements Rust name-lookup semantics in expressions, it will sometimes prepend the current crate to a name. For example, if GDB is stopped at a breakpoint in the crate '`K`', then `print ::x::y` will try to find the symbol '`K::x::y`'.

  However, since it is useful to be able to refer to other crates when debugging, GDB provides the `extern` extension to circumvent this. To use the extension, just put `extern` before a path expression to refer to the otherwise unavailable "global" scope.

  In the above example, if you wanted to refer to the symbol '`y`' in the crate '`x`', you would use `print extern x::y`.

- The Rust expression evaluator does not support "statement-like" expressions such as `if` or `match`, or lambda expressions.

- Tuple expressions are not implemented.
- The Rust expression evaluator does not currently implement the `Drop` trait. Objects that may be created by the evaluator will never be destroyed.
- GDB does not implement type inference for generics. In order to call generic functions or otherwise refer to generic items, you will have to specify the type parameters manually.
- GDB currently uses the C++ demangler for Rust. In most cases this does not cause any problems. However, in an expression context, completing a generic function name will give syntactically invalid results. This happens because Rust requires the '`::`' operator between the function name and its generic arguments. For example, GDB might provide a completion like `crate::f<u32>`, where the parser would require `crate::f::<u32>`.
- As of this writing, the Rust compiler (version 1.8) has a few holes in the debugging information it generates. These holes prevent certain features from being implemented by GDB:
  - Method calls cannot be made via traits.
  - Operator overloading is not implemented.
  - When debugging in a monomorphized function, you cannot use the generic type names.
  - The type `Self` is not available.
  - `use` statements are not available, so some names may not be available in the crate.

15.4.9 Modula-2

The extensions made to GDB to support Modula-2 only support output from the GNU Modula-2 compiler (which is currently being developed). Other Modula-2 compilers are not currently supported, and attempting to debug executables produced by them is most likely to give an error as GDB reads in the executable's symbol table.

15.4.9.1 Operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on structures. Operators are often defined on groups of types. For the purposes of Modula-2, the following definitions hold:

- *Integral types* consist of `INTEGER`, `CARDINAL`, and their subranges.
- *Character types* consist of `CHAR` and its subranges.
- *Floating-point types* consist of `REAL`.
- *Pointer types* consist of anything declared as `POINTER TO` *type*.
- *Scalar types* consist of all of the above.
- *Set types* consist of `SET` and `BITSET` types.
- *Boolean types* consist of `BOOLEAN`.

The following operators are supported, and appear in order of increasing precedence:

`,` Function argument or array index separator.

`:=` Assignment. The value of *var* `:=` *value* is *value*.

`<`, `>` Less than, greater than on integral, floating-point, or enumerated types.

`<=`, `>=` Less than or equal to, greater than or equal to on integral, floating-point and enumerated types, or set inclusion on set types. Same precedence as `<`.

`=`, `<>`, `#` Equality and two ways of expressing inequality, valid on scalar types. Same precedence as `<`. In GDB scripts, only `<>` is available for inequality, since `#` conflicts with the script comment character.

`IN` Set membership. Defined on set types and the types of their members. Same precedence as `<`.

`OR` Boolean disjunction. Defined on boolean types.

`AND`, `&` Boolean conjunction. Defined on boolean types.

`@` The GDB "artificial array" operator (see Section 10.1 [Expressions], page 137).

`+`, `-` Addition and subtraction on integral and floating-point types, or union and difference on set types.

`*` Multiplication on integral and floating-point types, or set intersection on set types.

`/` Division on floating-point types, or symmetric set difference on set types. Same precedence as `*`.

`DIV`, `MOD` Integer division and remainder. Defined on integral types. Same precedence as `*`.

`-` Negative. Defined on `INTEGER` and `REAL` data.

`^` Pointer dereferencing. Defined on pointer types.

`NOT` Boolean negation. Defined on boolean types. Same precedence as `^`.

`.` `RECORD` field selector. Defined on `RECORD` data. Same precedence as `^`.

`[]` Array indexing. Defined on `ARRAY` data. Same precedence as `^`.

`()` Procedure argument list. Defined on `PROCEDURE` objects. Same precedence as `^`.

`::`, `.` GDB and Modula-2 scope operators.

Warning: Set expressions and their operations are not yet supported, so GDB treats the use of the operator `IN`, or the use of operators `+`, `-`, `*`, `/`, `=`, , `<>`, `#`, `<=`, and `>=` on sets as an error.

15.4.9.2 Built-in Functions and Procedures

Modula-2 also makes available several built-in procedures and functions. In describing these, the following metavariables are used:

a represents an `ARRAY` variable.

c represents a `CHAR` constant or variable.

i represents a variable or constant of integral type.

m represents an identifier that belongs to a set. Generally used in the same function with the metavariable *s*. The type of *s* should be `SET OF` *mtype* (where *mtype* is the type of *m*).

n represents a variable or constant of integral or floating-point type.

r represents a variable or constant of floating-point type.

t represents a type.

v represents a variable.

x represents a variable or constant of one of many types. See the explanation of the function for details.

All Modula-2 built-in procedures also return a result, described below.

`ABS(n)` Returns the absolute value of *n*.

`CAP(c)` If *c* is a lower case letter, it returns its upper case equivalent, otherwise it returns its argument.

`CHR(i)` Returns the character whose ordinal value is *i*.

`DEC(v)` Decrements the value in the variable *v* by one. Returns the new value.

`DEC(v,i)` Decrements the value in the variable *v* by *i*. Returns the new value.

`EXCL(m,s)`
Removes the element *m* from the set *s*. Returns the new set.

`FLOAT(i)` Returns the floating point equivalent of the integer *i*.

`HIGH(a)` Returns the index of the last member of *a*.

`INC(v)` Increments the value in the variable *v* by one. Returns the new value.

`INC(v,i)` Increments the value in the variable *v* by *i*. Returns the new value.

`INCL(m,s)`
Adds the element *m* to the set *s* if it is not already there. Returns the new set.

`MAX(t)` Returns the maximum value of the type *t*.

`MIN(t)` Returns the minimum value of the type *t*.

`ODD(i)` Returns boolean TRUE if *i* is an odd number.

`ORD(x)` Returns the ordinal value of its argument. For example, the ordinal value of a character is its ASCII value (on machines supporting the ASCII character set). The argument *x* must be of an ordered type, which include integral, character and enumerated types.

`SIZE(x)` Returns the size of its argument. The argument *x* can be a variable or a type.

`TRUNC(r)` Returns the integral part of *r*.

`TSIZE(x)` Returns the size of its argument. The argument *x* can be a variable or a type.

`VAL(t,i)` Returns the member of the type *t* whose ordinal value is *i*.

Warning: Sets and their operations are not yet supported, so GDB treats the use of procedures `INCL` and `EXCL` as an error.

15.4.9.3 Constants

GDB allows you to express the constants of Modula-2 in the following ways:

- Integer constants are simply a sequence of digits. When used in an expression, a constant is interpreted to be type-compatible with the rest of the expression. Hexadecimal integers are specified by a trailing 'H', and octal integers by a trailing 'B'.
- Floating point constants appear as a sequence of digits, followed by a decimal point and another sequence of digits. An optional exponent can then be specified, in the form 'E[+|-]*nnn*', where '[+|-]*nnn*' is the desired exponent. All of the digits of the floating point constant must be valid decimal (base 10) digits.
- Character constants consist of a single character enclosed by a pair of like quotes, either single (') or double ("). They may also be expressed by their ordinal value (their ASCII value, usually) followed by a 'C'.
- String constants consist of a sequence of characters enclosed by a pair of like quotes, either single (') or double ("). Escape sequences in the style of C are also allowed. See Section 15.4.1.2 [C and C++ Constants], page 223, for a brief explanation of escape sequences.
- Enumerated constants consist of an enumerated identifier.
- Boolean constants consist of the identifiers TRUE and FALSE.
- Pointer constants consist of integral values only.
- Set constants are not yet supported.

15.4.9.4 Modula-2 Types

Currently GDB can print the following data types in Modula-2 syntax: array types, record types, set types, pointer types, procedure types, enumerated types, subrange types and base types. You can also print the contents of variables declared using these type. This section gives a number of simple source code examples together with sample GDB sessions.

The first example contains the following section of code:

```
VAR
   s: SET OF CHAR ;
   r: [20..40] ;
```

and you can request GDB to interrogate the type and value of `r` and `s`.

```
(gdb) print s
{'A'..'C', 'Z'}
(gdb) ptype s
SET OF CHAR
(gdb) print r
21
(gdb) ptype r
[20..40]
```

Likewise if your source code declares `s` as:

```
VAR
   s: SET ['A'..'Z'] ;
```

then you may query the type of `s` by:

```
(gdb) ptype s
type = SET ['A'..'Z']
```

Note that at present you cannot interactively manipulate set expressions using the debugger.

The following example shows how you might declare an array in Modula-2 and how you can interact with GDB to print its type and contents:

```
VAR
   s: ARRAY [-10..10] OF CHAR ;
(gdb) ptype s
ARRAY [-10..10] OF CHAR
```

Note that the array handling is not yet complete and although the type is printed correctly, expression handling still assumes that all arrays have a lower bound of zero and not `-10` as in the example above.

Here are some more type related Modula-2 examples:

```
TYPE
   colour = (blue, red, yellow, green) ;
   t = [blue..yellow] ;
VAR
   s: t ;
BEGIN
   s := blue ;
```

The GDB interaction shows how you can query the data type and value of a variable.

```
(gdb) print s
$1 = blue
(gdb) ptype t
type = [blue..yellow]
```

In this example a Modula-2 array is declared and its contents displayed. Observe that the contents are written in the same way as their `C` counterparts.

```
VAR
   s: ARRAY [1..5] OF CARDINAL ;
BEGIN
   s[1] := 1 ;
(gdb) print s
$1 = {1, 0, 0, 0, 0}
(gdb) ptype s
type = ARRAY [1..5] OF CARDINAL
```

The Modula-2 language interface to GDB also understands pointer types as shown in this example:

```
VAR
   s: POINTER TO ARRAY [1..5] OF CARDINAL ;
BEGIN
   NEW(s) ;
   s^[1] := 1 ;
```

and you can request that GDB describes the type of `s`.

```
(gdb) ptype s
type = POINTER TO ARRAY [1..5] OF CARDINAL
```

GDB handles compound types as we can see in this example. Here we combine array types, record types, pointer types and subrange types:

```
TYPE
   foo = RECORD
            f1: CARDINAL ;
            f2: CHAR ;
            f3: myarray ;
         END ;
```

```
   myarray = ARRAY myrange OF CARDINAL ;
   myrange = [-2..2] ;
VAR
   s: POINTER TO ARRAY myrange OF foo ;
```

and you can ask GDB to describe the type of `s` as shown below.

```
(gdb) ptype s
type = POINTER TO ARRAY [-2..2] OF foo = RECORD
    f1 : CARDINAL;
    f2 : CHAR;
    f3 : ARRAY [-2..2] OF CARDINAL;
END
```

15.4.9.5 Modula-2 Defaults

If type and range checking are set automatically by GDB, they both default to `on` whenever the working language changes to Modula-2. This happens regardless of whether you or GDB selected the working language.

If you allow GDB to set the language automatically, then entering code compiled from a file whose name ends with `.mod` sets the working language to Modula-2. See Section 15.1.3 [Having GDB Infer the Source Language], page 218, for further details.

15.4.9.6 Deviations from Standard Modula-2

A few changes have been made to make Modula-2 programs easier to debug. This is done primarily via loosening its type strictness:

- Unlike in standard Modula-2, pointer constants can be formed by integers. This allows you to modify pointer variables during debugging. (In standard Modula-2, the actual address contained in a pointer variable is hidden from you; it can only be modified through direct assignment to another pointer variable or expression that returned a pointer.)
- C escape sequences can be used in strings and characters to represent non-printable characters. GDB prints out strings with these escape sequences embedded. Single non-printable characters are printed using the '`CHR(`*nnn*`)`' format.
- The assignment operator (`:=`) returns the value of its right-hand argument.
- All built-in procedures both modify *and* return their argument.

15.4.9.7 Modula-2 Type and Range Checks

Warning: in this release, GDB does not yet perform type or range checking.

GDB considers two Modula-2 variables type equivalent if:

- They are of types that have been declared equivalent via a `TYPE` *t1* `=` *t2* statement
- They have been declared on the same line. (Note: This is true of the GNU Modula-2 compiler, but it may not be true of other compilers.)

As long as type checking is enabled, any attempt to combine variables whose types are not equivalent is an error.

Range checking is done on all mathematical operations, assignment, array index bounds, and all built-in functions and procedures.

15.4.9.8 The Scope Operators :: and .

There are a few subtle differences between the Modula-2 scope operator (`.`) and the GDB scope operator (`::`). The two have similar syntax:

```
module . id
scope :: id
```

where *scope* is the name of a module or a procedure, *module* the name of a module, and *id* is any declared identifier within your program, except another module.

Using the `::` operator makes GDB search the scope specified by *scope* for the identifier *id*. If it is not found in the specified scope, then GDB searches all scopes enclosing the one specified by *scope*.

Using the `.` operator makes GDB search the current scope for the identifier specified by *id* that was imported from the definition module specified by *module*. With this operator, it is an error if the identifier *id* was not imported from definition module *module*, or if *id* is not an identifier in *module*.

15.4.9.9 GDB and Modula-2

Some GDB commands have little use when debugging Modula-2 programs. Five subcommands of `set print` and `show print` apply specifically to C and C++: '`vtbl`', '`demangle`', '`asm-demangle`', '`object`', and '`union`'. The first four apply to C++, and the last to the C `union` type, which has no direct analogue in Modula-2.

The `@` operator (see Section 10.1 [Expressions], page 137), while available with any language, is not useful with Modula-2. Its intent is to aid the debugging of *dynamic arrays*, which cannot be created in Modula-2 as they can in C or C++. However, because an address can be specified by an integral constant, the construct '`{type}adrexp`' is still useful.

In GDB scripts, the Modula-2 inequality operator `#` is interpreted as the beginning of a comment. Use `<>` instead.

15.4.10 Ada

The extensions made to GDB for Ada only support output from the GNU Ada (GNAT) compiler. Other Ada compilers are not currently supported, and attempting to debug executables produced by them is most likely to be difficult.

15.4.10.1 Introduction

The Ada mode of GDB supports a fairly large subset of Ada expression syntax, with some extensions. The philosophy behind the design of this subset is

- That GDB should provide basic literals and access to operations for arithmetic, dereferencing, field selection, indexing, and subprogram calls, leaving more sophisticated computations to subprograms written into the program (which therefore may be called from GDB).
- That type safety and strict adherence to Ada language restrictions are not particularly important to the GDB user.
- That brevity is important to the GDB user.

Thus, for brevity, the debugger acts as if all names declared in user-written packages are directly visible, even if they are not visible according to Ada rules, thus making it

unnecessary to fully qualify most names with their packages, regardless of context. Where this causes ambiguity, GDB asks the user's intent.

The debugger will start in Ada mode if it detects an Ada main program. As for other languages, it will enter Ada mode when stopped in a program that was translated from an Ada source file.

While in Ada mode, you may use '`--`' for comments. This is useful mostly for documenting command files. The standard GDB comment ('`#`') still works at the beginning of a line in Ada mode, but not in the middle (to allow based literals).

15.4.10.2 Omissions from Ada

Here are the notable omissions from the subset:

- Only a subset of the attributes are supported:
  - `'First`, `'Last`, and `'Length` on array objects (not on types and subtypes).
  - `'Min` and `'Max`.
  - `'Pos` and `'Val`.
  - `'Tag`.
  - `'Range` on array objects (not subtypes), but only as the right operand of the membership (`in`) operator.
  - `'Access`, `'Unchecked_Access`, and `'Unrestricted_Access` (a GNAT extension).
  - `'Address`.
- The names in `Characters.Latin_1` are not available.
- Equality tests ('`=`' and '`/=`') on arrays test for bitwise equality of representations. They will generally work correctly for strings and arrays whose elements have integer or enumeration types. They may not work correctly for arrays whose element types have user-defined equality, for arrays of real values (in particular, IEEE-conformant floating point, because of negative zeroes and NaNs), and for arrays whose elements contain unused bits with indeterminate values.
- The other component-by-component array operations (`and`, `or`, `xor`, `not`, and relational tests other than equality) are not implemented.
- There is limited support for array and record aggregates. They are permitted only on the right sides of assignments, as in these examples:

```
(gdb) set An_Array := (1, 2, 3, 4, 5, 6)
(gdb) set An_Array := (1, others => 0)
(gdb) set An_Array := (0|4 => 1, 1..3 => 2, 5 => 6)
(gdb) set A_2D_Array := ((1, 2, 3), (4, 5, 6), (7, 8, 9))
(gdb) set A_Record := (1, "Peter", True);
(gdb) set A_Record := (Name => "Peter", Id => 1, Alive => True)
```

  Changing a discriminant's value by assigning an aggregate has an undefined effect if that discriminant is used within the record. However, you can first modify discriminants by directly assigning to them (which normally would not be allowed in Ada), and then performing an aggregate assignment. For example, given a variable `A_Rec` declared to have a type such as:

```
type Rec (Len : Small_Integer := 0) is record
    Id : Integer;
    Vals : IntArray (1 .. Len);
```

```
end record;
```

you can assign a value with a different size of `Vals` with two assignments:

```
(gdb) set A_Rec.Len := 4
(gdb) set A_Rec := (Id => 42, Vals => (1, 2, 3, 4))
```

As this example also illustrates, GDB is very loose about the usual rules concerning aggregates. You may leave out some of the components of an array or record aggregate (such as the `Len` component in the assignment to `A_Rec` above); they will retain their original values upon assignment. You may freely use dynamic values as indices in component associations. You may even use overlapping or redundant component associations, although which component values are assigned in such cases is not defined.

- Calls to dispatching subprograms are not implemented.
- The overloading algorithm is much more limited (i.e., less selective) than that of real Ada. It makes only limited use of the context in which a subexpression appears to resolve its meaning, and it is much looser in its rules for allowing type matches. As a result, some function calls will be ambiguous, and the user will be asked to choose the proper resolution.
- The `new` operator is not implemented.
- Entry calls are not implemented.
- Aside from printing, arithmetic operations on the native VAX floating-point formats are not supported.
- It is not possible to slice a packed array.
- The names `True` and `False`, when not part of a qualified name, are interpreted as if implicitly prefixed by `Standard`, regardless of context. Should your program redefine these names in a package or procedure (at best a dubious practice), you will have to use fully qualified names to access their new definitions.
- Based real literals are not implemented.

15.4.10.3 Additions to Ada

As it does for other languages, GDB makes certain generic extensions to Ada (see Section 10.1 [Expressions], page 137):

- If the expression E is a variable residing in memory (typically a local variable or array element) and N is a positive integer, then *`E@N`* displays the values of E and the N-1 adjacent variables following it in memory as an array. In Ada, this operator is generally not necessary, since its prime use is in displaying parts of an array, and slicing will usually do this in Ada. However, there are occasional uses when debugging programs in which certain debugging information has been optimized away.
- *`B::var`* means "the variable named *var* that appears in function or file *B*." When *B* is a file name, you must typically surround it in single quotes.
- The expression *`{type} addr`* means "the variable of type *type* that appears at address *addr*."
- A name starting with '`$`' is a convenience variable (see Section 10.12 [Convenience Vars], page 162) or a machine register (see Section 10.14 [Registers], page 169).

In addition, GDB provides a few other shortcuts and outright additions specific to Ada:

- The assignment statement is allowed as an expression, returning its right-hand operand as its value. Thus, you may enter

```
(gdb) set x := y + 3
(gdb) print A(tmp := y + 1)
```

- The semicolon is allowed as an "operator," returning as its value the value of its right-hand operand. This allows, for example, complex conditional breaks:

```
(gdb) break f
(gdb) condition 1 (report(i); k += 1; A(k) > 100)
```

- An extension to based literals can be used to specify the exact byte contents of a floating-point literal. After the base, you can use from zero to two '`l`' characters, followed by an '`f`'. The number of '`l`' characters controls the width of the resulting real constant: zero means `Float` is used, one means `Long_Float`, and two means `Long_Long_Float`.

```
(gdb) print 16f#41b80000#
$1 = 23.0
```

- Rather than use catenation and symbolic character names to introduce special characters into strings, one may instead use a special bracket notation, which is also used to print strings. A sequence of characters of the form '`["`*XX*`"]`' within a string or character literal denotes the (single) character whose numeric encoding is *XX* in hexadecimal. The sequence of characters '`["""]`' also denotes a single quotation mark in strings. For example,

```
"One line.["0a"]Next line.["0a"]"
```

  contains an ASCII newline character (`Ada.Characters.Latin_1.LF`) after each period.

- The subtype used as a prefix for the attributes `'Pos`, `'Min`, and `'Max` is optional (and is ignored in any case). For example, it is valid to write

```
(gdb) print 'max(x, y)
```

- When printing arrays, GDB uses positional notation when the array has a lower bound of 1, and uses a modified named notation otherwise. For example, a one-dimensional array of three integers with a lower bound of 3 might print as

```
(3 => 10, 17, 1)
```

  That is, in contrast to valid Ada, only the first component has a `=>` clause.

- You may abbreviate attributes in expressions with any unique, multi-character subsequence of their names (an exact match gets preference). For example, you may use `a'len`, `a'gth`, or `a'lh` in place of `a'length`.

- Since Ada is case-insensitive, the debugger normally maps identifiers you type to lower case. The GNAT compiler uses upper-case characters for some of its internal identifiers, which are normally of no interest to users. For the rare occasions when you actually have to look at them, enclose them in angle brackets to avoid the lower-case mapping. For example,

```
(gdb) print <JMPBUF_SAVE>[0]
```

- Printing an object of class-wide type or dereferencing an access-to-class-wide value will display all the components of the object's specific type (as indicated by its run-time tag). Likewise, component selection on such a value will operate on the specific type of the object.

15.4.10.4 Overloading support for Ada

The debugger supports limited overloading. Given a subprogram call in which the function symbol has multiple definitions, it will use the number of actual parameters and some information about their types to attempt to narrow the set of definitions. It also makes very limited use of context, preferring procedures to functions in the context of the `call` command, and functions to procedures elsewhere.

If, after narrowing, the set of matching definitions still contains more than one definition, GDB will display a menu to query which one it should use, for instance:

```
(gdb) print f(1)
Multiple matches for f
[0] cancel
[1] foo.f (integer) return boolean at foo.adb:23
[2] foo.f (foo.new_integer) return boolean at foo.adb:28
>
```

In this case, just select one menu entry either to cancel expression evaluation (type *0* and press `RET`) or to continue evaluation with a specific instance (type the corresponding number and press `RET`).

Here are a couple of commands to customize GDB's behavior in this case:

`set ada print-signatures`
: Control whether parameter types and return types are displayed in overloads selection menus. It is `on` by default. See Section 15.4.10.4 [Overloading support for Ada], page 242.

`show ada print-signatures`
: Show the current setting for displaying parameter types and return types in overloads selection menu. See Section 15.4.10.4 [Overloading support for Ada], page 242.

15.4.10.5 Stopping at the Very Beginning

It is sometimes necessary to debug the program during elaboration, and before reaching the main procedure. As defined in the Ada Reference Manual, the elaboration code is invoked from a procedure called `adainit`. To run your program up to the beginning of elaboration, simply use the following two commands: `tbreak adainit` and `run`.

15.4.10.6 Ada Exceptions

A command is provided to list all Ada exceptions:

`info exceptions`
`info exceptions` *regexp*
: The `info exceptions` command allows you to list all Ada exceptions defined within the program being debugged, as well as their addresses. With a regular expression, *regexp*, as argument, only those exceptions whose names match *regexp* are listed.

Below is a small example, showing how the command can be used, first without argument, and next with a regular expression passed as an argument.

```
(gdb) info exceptions
All defined Ada exceptions:
```

```
constraint_error: 0x613da0
program_error: 0x613d20
storage_error: 0x613ce0
tasking_error: 0x613ca0
const.aint_global_e: 0x613b00
(gdb) info exceptions const.aint
All Ada exceptions matching regular expression "const.aint":
constraint_error: 0x613da0
const.aint_global_e: 0x613b00
```

It is also possible to ask GDB to stop your program's execution when an exception is raised. For more details, see Section 5.1.3 [Set Catchpoints], page 66.

15.4.10.7 Extensions for Ada Tasks

Support for Ada tasks is analogous to that for threads (see Section 4.10 [Threads], page 44). GDB provides the following task-related commands:

`info tasks`

This command shows a list of current Ada tasks, as in the following example:

```
(gdb) info tasks
  ID       TID P-ID Pri State                  Name
   1   8088000   0   15 Child Activation Wait main_task
   2   80a4000   1   15 Accept Statement      b
   3   809a800   1   15 Child Activation Wait a
*  4   80ae800   3   15 Runnable             c
```

In this listing, the asterisk before the last task indicates it to be the task currently being inspected.

ID — Represents GDB's internal task number.

TID — The Ada task ID.

P-ID — The parent's task ID (GDB's internal task number).

Pri — The base priority of the task.

State — Current state of the task.

`Unactivated`
: The task has been created but has not been activated. It cannot be executing.

`Runnable`
: The task is not blocked for any reason known to Ada. (It may be waiting for a mutex, though.) It is conceptually "executing" in normal mode.

`Terminated`
: The task is terminated, in the sense of ARM 9.3 (5). Any dependents that were waiting on terminate alternatives have been awakened and have terminated themselves.

`Child Activation Wait`
: The task is waiting for created tasks to complete activation.

`Accept Statement`
: The task is waiting on an accept or selective wait statement.

`Waiting on entry call`
: The task is waiting on an entry call.

`Async Select Wait`
: The task is waiting to start the abortable part of an asynchronous select statement.

`Delay Sleep`
: The task is waiting on a select statement with only a delay alternative open.

`Child Termination Wait`
: The task is sleeping having completed a master within itself, and is waiting for the tasks dependent on that master to become terminated or waiting on a terminate Phase.

`Wait Child in Term Alt`
: The task is sleeping waiting for tasks on terminate alternatives to finish terminating.

`Accepting RV with` *taskno*
: The task is accepting a rendez-vous with the task *taskno*.

Name
: Name of the task in the program.

`info task` *taskno*
: This command shows detailed informations on the specified task, as in the following example:

```
(gdb) info tasks
  ID       TID P-ID Pri State                  Name
   1   8077880    0  15 Child Activation Wait  main_task
*  2   807c468    1  15 Runnable               task_1
(gdb) info task 2
Ada Task: 0x807c468
Name: "task_1"
Thread: 0
LWP: 0x1fac
Parent: 1 ("main_task")
Base Priority: 15
State: Runnable
```

`task`
: This command prints the ID and name of the current task.

```
(gdb) info tasks
  ID       TID P-ID Pri State                  Name
   1   8077870    0  15 Child Activation Wait  main_task
*  2   807c458    1  15 Runnable               some_task
(gdb) task
[Current task is 2 "some_task"]
```

`task` *taskno*

This command is like the `thread` *thread-id* command (see Section 4.10 [Threads], page 44). It switches the context of debugging from the current task to the given task.

```
(gdb) info tasks
  ID       TID P-ID Pri State                  Name
   1   8077870    0  15 Child Activation Wait  main_task
*  2   807c458    1  15 Runnable               some_task
(gdb) task 1
[Switching to task 1 "main_task"]
#0  0x8067726 in pthread_cond_wait ()
(gdb) bt
#0  0x8067726 in pthread_cond_wait ()
#1  0x8056714 in system.os_interface.pthread_cond_wait ()
#2  0x805cb63 in system.task_primitives.operations.sleep ()
#3  0x806153e in system.tasking.stages.activate_tasks ()
#4  0x804aacc in un () at un.adb:5
```

`task apply` [*task-id-list* | `all`] [*flag*]... *command*

The `task apply` command is the Ada tasking analogue of `thread apply` (see Section 4.10 [Threads], page 44). It allows you to apply the named *command* to one or more tasks. Specify the tasks that you want affected using a list of task IDs, or specify `all` to apply to all tasks.

The *flag* arguments control what output to produce and how to handle errors raised when applying *command* to a task. *flag* must start with a `-` directly followed by one letter in `qcs`. If several flags are provided, they must be given individually, such as `-c -q`.

By default, GDB displays some task information before the output produced by *command*, and an error raised during the execution of a *command* will abort `task apply`. The following flags can be used to fine-tune this behavior:

`-c` The flag `-c`, which stands for '`continue`', causes any errors in *command* to be displayed, and the execution of `task apply` then continues.

`-s` The flag `-s`, which stands for '`silent`', causes any errors or empty output produced by a *command* to be silently ignored. That is, the execution continues, but the task information and errors are not printed.

`-q` The flag `-q` ('`quiet`') disables printing the task information.

Flags `-c` and `-s` cannot be used together.

`break` *location* `task` *taskno*
`break` *location* `task` *taskno* `if ...`

These commands are like the `break ... thread ...` command (see Section 5.5 [Thread Stops], page 89). The *location* argument specifies source lines, as described in Section 9.2 [Specify Location], page 120.

Use the qualifier '`task` *taskno*' with a breakpoint command to specify that you only want GDB to stop the program when a particular Ada task reaches this breakpoint. The *taskno* is one of the numeric task identifiers assigned by GDB, shown in the first column of the '`info tasks`' display.

If you do not specify '`task` *taskno*' when you set a breakpoint, the breakpoint applies to *all* tasks of your program.

You can use the `task` qualifier on conditional breakpoints as well; in this case, place '`task` *taskno*' before the breakpoint condition (before the `if`).

For example,

```
(gdb) info tasks
  ID       TID P-ID Pri State                 Name
   1 140022020   0   15 Child Activation Wait main_task
   2 140045060   1   15 Accept/Select Wait    t2
   3 140044840   1   15 Runnable              t1
*  4 140056040   1   15 Runnable              t3
(gdb) b 15 task 2
Breakpoint 5 at 0x120044cb0: file test_task_debug.adb, line 15.
(gdb) cont
Continuing.
task # 1 running
task # 2 running

Breakpoint 5, test_task_debug () at test_task_debug.adb:15
15               flush;
(gdb) info tasks
  ID       TID P-ID Pri State                 Name
   1 140022020   0   15 Child Activation Wait main_task
*  2 140045060   1   15 Runnable              t2
   3 140044840   1   15 Runnable              t1
   4 140056040   1   15 Delay Sleep           t3
```

15.4.10.8 Tasking Support when Debugging Core Files

When inspecting a core file, as opposed to debugging a live program, tasking support may be limited or even unavailable, depending on the platform being used. For instance, on x86-linux, the list of tasks is available, but task switching is not supported.

On certain platforms, the debugger needs to perform some memory writes in order to provide Ada tasking support. When inspecting a core file, this means that the core file must be opened with read-write privileges, using the command '`"set write on"`' (see Section 17.6 [Patching], page 271). Under these circumstances, you should make a backup copy of the core file before inspecting it with GDB.

15.4.10.9 Tasking Support when using the Ravenscar Profile

The *Ravenscar Profile* is a subset of the Ada tasking features, specifically designed for systems with safety-critical real-time requirements.

`set ravenscar task-switching on`
: Allows task switching when debugging a program that uses the Ravenscar Profile. This is the default.

`set ravenscar task-switching off`
: Turn off task switching when debugging a program that uses the Ravenscar Profile. This is mostly intended to disable the code that adds support for the Ravenscar Profile, in case a bug in either GDB or in the Ravenscar runtime is preventing GDB from working properly. To be effective, this command should be run before the program is started.

`show ravenscar task-switching`
: Show whether it is possible to switch from task to task in a program using the Ravenscar Profile.

When Ravenscar task-switching is enabled, Ravenscar tasks are announced by GDB as if they were threads:

```
(gdb) continue
[New Ravenscar Thread 0x2b8f0]
```

Both Ravenscar tasks and the underlying CPU threads will show up in the output of `info threads`:

```
(gdb) info threads
  Id   Target Id                  Frame
  1    Thread 1 (CPU#0 [running]) simple () at simple.adb:10
  2    Thread 2 (CPU#1 [running]) 0x0000000000003d34 in __gnat_initialize_cpu_devices ()
  3    Thread 3 (CPU#2 [running]) 0x0000000000003d28 in __gnat_initialize_cpu_devices ()
  4    Thread 4 (CPU#3 [halted ]) 0x000000000000c6ec in system.task_primitives.operations.idle ()
* 5    Ravenscar Thread 0x2b8f0   simple () at simple.adb:10
  6    Ravenscar Thread 0x2f150   0x000000000000c6ec in system.task_primitives.operations.idle ()
```

One known limitation of the Ravenscar support in GDB is that it isn't currently possible to single-step through the runtime initialization sequence. If you need to debug this code, you should use `set ravenscar task-switching off`.

15.4.10.10 Ada Source Character Set

The GNAT compiler supports a number of character sets for source files. See Section "Character Set Control" in `gnat_ugn`. GDB includes support for this as well.

`set ada source-charset` *`charset`*
: Set the source character set for Ada. The character set must be supported by GNAT. Because this setting affects the decoding of symbols coming from the debug information in your program, the setting should be set as early as possible. The default is `ISO-8859-1`, because that is also GNAT's default.

`show ada source-charset`
: Show the current source character set for Ada.

15.4.10.11 Known Peculiarities of Ada Mode

Besides the omissions listed previously (see Section 15.4.10.2 [Omissions from Ada], page 239), we know of several problems with and limitations of Ada mode in GDB, some of which will be fixed with planned future releases of the debugger and the GNU Ada compiler.

- Static constants that the compiler chooses not to materialize as objects in storage are invisible to the debugger.
- Named parameter associations in function argument lists are ignored (the argument lists are treated as positional).
- Many useful library packages are currently invisible to the debugger.
- Fixed-point arithmetic, conversions, input, and output is carried out using floating-point arithmetic, and may give results that only approximate those on the host machine.
- The GNAT compiler never generates the prefix `Standard` for any of the standard symbols defined by the Ada language. GDB knows about this: it will strip the prefix from names when you use it, and will never look for a name you have so qualified among local symbols, nor match against symbols in other packages or subprograms. If you have defined entities anywhere in your program other than parameters and local variables whose simple names match names in `Standard`, GNAT's lack of qualification here can cause confusion. When this happens, you can usually resolve the confusion by qualifying the problematic names with package `Standard` explicitly.

Older versions of the compiler sometimes generate erroneous debugging information, resulting in the debugger incorrectly printing the value of affected entities. In some cases, the debugger is able to work around an issue automatically. In other cases, the debugger is able to work around the issue, but the work-around has to be specifically enabled.

`set ada trust-PAD-over-XVS on`
: Configure GDB to strictly follow the GNAT encoding when computing the value of Ada entities, particularly when `PAD` and `PAD___XVS` types are involved (see `ada/exp_dbug.ads` in the GCC sources for a complete description of the encoding used by the GNAT compiler). This is the default.

`set ada trust-PAD-over-XVS off`
: This is related to the encoding using by the GNAT compiler. If GDB sometimes prints the wrong value for certain entities, changing `ada trust-PAD-over-XVS` to `off` activates a work-around which may fix the issue. It is always safe to set `ada trust-PAD-over-XVS` to `off`, but this incurs a slight performance penalty, so it is recommended to leave this setting to `on` unless necessary.

Internally, the debugger also relies on the compiler following a number of conventions known as the '`GNAT Encoding`', all documented in `gcc/ada/exp_dbug.ads` in the GCC sources. This encoding describes how the debugging information should be generated for certain types. In particular, this convention makes use of *descriptive types*, which are artificial types generated purely to help the debugger.

These encodings were defined at a time when the debugging information format used was not powerful enough to describe some of the more complex types available in Ada. Since DWARF allows us to express nearly all Ada features, the long-term goal is to slowly replace

these descriptive types by their pure DWARF equivalent. To facilitate that transition, a new maintenance option is available to force the debugger to ignore those descriptive types. It allows the user to quickly evaluate how well GDB works without them.

`maintenance ada set ignore-descriptive-types [on|off]`
: Control whether the debugger should ignore descriptive types. The default is not to ignore descriptives types (`off`).

`maintenance ada show ignore-descriptive-types`
: Show if descriptive types are ignored by GDB.

15.5 Unsupported Languages

In addition to the other fully-supported programming languages, GDB also provides a pseudo-language, called `minimal`. It does not represent a real programming language, but provides a set of capabilities close to what the C or assembly languages provide. This should allow most simple operations to be performed while debugging an application that uses a language currently not supported by GDB.

If the language is set to `auto`, GDB will automatically select this language if the current frame corresponds to an unsupported language.

16 Examining the Symbol Table

The commands described in this chapter allow you to inquire about the symbols (names of variables, functions and types) de ned in your program. This information is inherent in the text of your program and does not change as your program executes. gdb nds it in your program's symbol table, in the le indicated when you started gdb (see Section 2.1.1 [Choosing Files], page 12), or by one of the le-management commands (see Section 18.1 [Commands to Specify Files], page 277).

Occasionally, you may need to refer to symbols that contain unusual characters, which gdb ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source les (see Section 10.3 [Program Variables], page 138). File names are recorded in object les as debugging symbols, but gdb would ordinarily parse a typical le name, like foo.c , as the three words foo ' `. ' `c'. To allow gdb to recognize foo.c ' as a single symbol, enclose it in single quotes; for example,

```
p 'foo.c'::x
```

looks up the value of x in the scope of the le foo.c .

set case-sensitive on
set case-sensitive off
set case-sensitive auto
: Normally, when gdb looks up symbols, it matches their names with case sensitivity determined by the current source language. Occasionally, you may wish to control that. The command set case-sensitive lets you do that by specifying on for case-sensitive matches or off for case-insensitive ones. If you specify auto , case sensitivity is reset to the default suitable for the source language. The default is case-sensitive matches for all languages except for Fortran, for which the default is case-insensitive matches.

show case-sensitive
: This command shows the current setting of case sensitivity for symbols lookups.

set print type methods
set print type methods on
set print type methods off
: Normally, when gdb prints a class, it displays any methods declared in that class. You can control this behavior either by passing the appropriate ag to ptype , or using set print type methods . Specifying on will cause gdb to display the methods; this is the default. Specifying off will cause gdb to omit the methods.

show print type methods
: This command shows the current setting of method display when printing classes.

set print type nested-type-limit limit
set print type nested-type-limit unlimited
: Set the limit of displayed nested types that the type printer will show. A limit of unlimited or -1 will show all nested de nitions. By default, the type printer will not show any nested types de ned in classes.

(The syntax above will vary depending on the shell used.) This example will execute commands from the file `cmds`. All output and errors would be directed to `log`.

```
echo This is some text\n\
which is continued\n\
onto several lines.\n
```

produces the same output as

```
echo This is some text\n
echo which is continued\n
echo onto several lines.\n
```

`output` *expression*

Print the value of *expression* and nothing but that value: no newlines, no '`$`*nn* `=` '. The value is not entered in the value history either. See Section 10.1 [Expressions], page 137, for more information on expressions.

`output/`*fmt* *expression*

Print the value of *expression* in format *fmt*. You can use the same formats as for `print`. See Section 10.5 [Output Formats], page 142, for more information.

`printf` *template*`,` *expressions*`...`

Print the values of one or more *expressions* under the control of the string *template*. To print several values, make *expressions* be a comma-separated

Additionally, `printf`

23.2.1 Default Arguments

You can tell GDB to always prepend some default arguments to the list of arguments provided

23.3.2 Python API

You can get quick online help for GDB

frames it has to iterate over, and it may have to iterate through them all. This ends up duplicating effort as GDB performs this iteration when it prints the frames. If the filter can defer unwinding frames until frame decorators are executed, after the last filter has executed, it should. See Section 23.3.2.10 [Frame Decorator API], page 404, for more information on decorators. Also, there are examples for both frame decorators and filters in later chapters. See Section 23.3.2.11 [Writing a Frame Filter], page 407, for more information.

The Python dictionary `gdb.frame_filters` contains key/object pairings that comprise a frame filter. Frame filters in this dictionary are called `global` frame filters, and they are available when debugging all 758g 151 Td [hre cald

This method must return an object conforming to the Python iterator protocol. Each item in the iterator must be an object conforming to the frame decorator interface. If a frame filter does not wish to perform any operations on this iterator, it should return that iterator untouched.

This method is not optional. If it does not exist, GDB will raise and print an error.

`FrameFilter.name` [Variable]

The `name` attribute must be Python string which contains the name of the filter displayed by GDB (see Section 8.6 [Frame Filter Management], page 116). This attribute may contain any combination of letters or numbers. Care should be taken to ensure that it is unique. This attribute is mandatory.

`FrameFilter.enabled` [Variable]

The `enabled` attribute must be Python boolean. This attribute indicates to GDB whether the frame filter is enabled, and should be considered when frame filters are executed. If `enabled` is `True`, then the frame filter will be executed when any of the backtrace commands detailed earlier in this chapter are executed. If `enabled` is `False`, then the frame filter will not be executed. This attribute is mandatory.

`FrameFilter.priority` [Variable]

The `priority` attribute must be Python integer. This attribute controls the order of execution in relation to other frame filters. There are no imposed limits on the range of `priority` other than it must be a valid integer. The higher the `priority` attribute, the sooner the frame filter will be executed in relation to other frame filters. Although `priority` can be negative, it is recommended practice to assume zero is the lowest priority that a frame filter can be assigned. Frame filters that have the same priority are executed in unsorted order in that priority slot. This attribute is mandatory. 100 is a good default priority.

23.3.2.10 Decorating Frames

Frame decorators are sister objects to frame filters (see Section 23.3.2.9 [Frame Filter API], page 402). Frame decorators are applied by a frame filter and can only be used in conjunction with frame filters.

The purpose of a frame decorator is to customize the printed content of each `gdb.Frame` in commands where frame filters are executed. This concept is called decorating a frame. Frame decorators decorate a `gdb.Frame` with Python code contained within each API call. This separates the actual data contained in a `gdb.Frame` from the decorated data produced by a frame decorator. This abstraction is necessary to maintain integrity of the data contained in each `gdb.Frame`.

Frame decorators have a mandatory interface, defined below.

GDB already contains a frame decorator called `FrameDecorator`. This contains substantial amounts of boilerplate code to decorate the content of a `gdb.Frame`. It is recommended that other frame decorators inherit and extend this object, and only to override the methods needed.

`FrameDecorator` is defined in the Python module `gdb.FrameDecorator`, so your code can import it like:

```
from gdb.FrameDecorator import FrameDecorator
```

`None`, and the `argument` method returns a `gdb.Symbol`, GDB will look-up and print the value of the `gdb.Symbol` automatically.

A brief example:

```
class SymValueWrapper():

    def __init__(self, symbol, value):
        self.sym = symbol
        self.val = value

    def value(self):
        return self.val

    def symbol(self):
        return self.sym

class SomeFrameDecorator()
...
...
    def frame_args(self):
        args = []
        try:
            block = self.inferior_frame.block()
        except:
            return None

        # Iterate over all symbols in a block.  Only add
        # symbols that are arguments.
        for sym in block:
            if not sym.is_argument:
                continue
            args.append(SymValueWrapper(sym,None))

        # Add example synthetic argument.
        args.append(SymValueWrapper(``foo'', 42))

        return args
```

`FrameDecorator.frame_locals` (*self*) [Function]

This method must return an iterable or `None`. Returning an empty iterable, or `None` means frame local arguments will not be printed for this frame.

The object interface, the description of the various strategies for reading frame locals, and the example are largely similar to those described in the `frame_args` function, (see [The frame filter frame_args function], page 405). Below is a modified example:

```
class SomeFrameDecorator()
...
...
    def frame_locals(self):
        vars = []
        try:
            block = self.inferior_frame.block()
        except:
            return None

        # Iterate over all symbols in a block.  Add all
        # symbols, except arguments.
        for sym in block:
```

```
            if sym.is_argument:
                continue
            vars.append(SymValueWrapper(sym,None))

        # Add an example of a synthetic local variable.
        vars.append(SymValueWrapper(``bar'', 99))

        return vars
```

`FrameDecorator.inferior_frame` (*self*): [Function]

This method must return the underlying `gdb.Frame` that this frame decorator is decorating. GDB requires the underlying frame for internal frame information to determine how to print certain values when printing a frame.

23.3.2.11 Writing a Frame Filter

There are three basic elements that a frame filter must implement: it must correctly implement the documented interface (see Section 23.3.2.9 [Frame Filter API], page 402), it must register itself with GDB

The second step is registering the frame filter with the dictionary or dictionaries that the frame filter has interest in. As shown in the comments, this filter just registers itself with the global dictionary `gdb.frame_filters`. As noted earlier, `gdb.frame_filters` is a dictionary that is initialized in the `gdb` module when GDB starts. What dictionary a filter registers with is an important consideration. Generally, if a filter is specific to a set of code, it should be registered either in the `objfile` or `progspace` dictionaries as they are specific to the program currently loaded in GDB. The global dictionary is always present in GDB and is never unloaded. Any filters registered with the global dictionary will exist until GDB exits. To avoid filters that may conflict, it is generally better to register frame filters against the dictionaries that more closely align with the usage of the filter currently in question. See Section 23.3.3 [Python Auto-loading], page 465, for further information on auto-loading Python scripts.

GDB takes a hands-off approach to frame filter registration, therefore it is the frame filter's responsibility to ensure registration has occurred, and that any exceptions are handled appropriately. In particular, youIn

performance viewpoint, this is the most appropriate decision to make as it avoids duplicating the effort that the printing step would undertake anyway. Also, if there are many frame filters unwinding the stack during filtering, it can substantially delay the printing of the backtrace which will result in large memory usage, and a poor user experience.

```
class InlineFilter():

    def __init__(self):
        self.name = "InlinedFrameFilter"
        self.priority = 100
        self.enabled = True
        gdb.frame_filters[self.name] = self

    def filter(self, frame_iter):
        frame_iter = itertools.imap(InlinedFrameDecorator,
                                    frame_iter)
        return frame_iter
```

This frame filter is somewhat similar to the earlier example, except that the `filter` method applies a frame decorator object called `InlinedFrameDecorator` to each element in the iterator. The `imap` Python method is light-weight. It does not proactively iterate over the iterator, but rather creates a new iterator which wraps the existing one.

Below is the frame decorator for this example.

```
class InlinedFrameDecorator(FrameDecorator):

    def __init__(self, fobj):
        super(InlinedFrameDecorator, self).__init__(fobj)

    def function(self):
        frame = self.inferior_frame()
        name = str(frame.name())

        if frame.type() == gdb.INLINE_FRAME:
            name = name + " [inlined]"

        return name
```

This frame decorator only defines and overrides the `function` method. It lets the supplied `FrameDecorator`, which is shipped with GDB, perform the other work associated with printing this frame.

The combination of these two objects create this output from a backtrace:

```
#0  0x004004e0 in bar () at inline.c:11
#1  0x00400566 in max [inlined] (b=6, a=12) at inline.c:21
#2  0x00400566 in main () at inline.c:31
```

So in the case of this example, a frame decorator is applied to all frames, regardless of whether they may be inlined or not. As GDB iterates over the iterator produced by the frame filters, GDB executes each frame decorator which then makes a decision on what to print in the `function` callback. Using a strategy like this is a way to defer decisions on the frame content to printing time.

Eliding Frames

It might be that the above example is not desirable for representing inlined frames, and a hierarchical approach may be preferred. If we want to hierarchically represent frames, the `elided` frame decorator interface might be preferable.

This example approaches the issue with the `elided` method. This example is quite long, but very simplistic. It is out-of-scope for this section to write a complete example that comprehensively covers all approaches of finding and printing inlined frames. However, this example illustrates the approach an author might use.

This example comprises of three sections.

```
class InlineFrameFilter():

    def __init__(self):
        self.name = "InlinedFrameFilter"
        self.priority = 100
        self.enabled = True
        gdb.frame_filters[self.name] = self

    def filter(self, frame_iter):
        return ElidingInlineIterator(frame_iter)
```

This frame filter is very similar to the other examples. The only difference is this frame filter is wrapping the iterator provided to it (`frame_iter`) with a custom iterator called `ElidingInlineIterator`. This again defers actions to when GDB prints the backtrace, as the iterator is not traversed until printing.

The iterator for this example is as follows. It is in this section of the example where decisions are made on the content of the backtrace.

```
class ElidingInlineIterator:
    def __init__(self, ii):
        self.input_iterator = ii

    def __iter__(self):
        return self

    def next(self):
        frame = next(self.input_iterator)

        if frame.inferior_frame().type() != gdb.INLINE_FRAME:
            return frame

        try:
            eliding_frame = next(self.input_iterator)
        except StopIteration:
            return frame
        return ElidingFrameDecorator(eliding_frame, [frame])
```

This iterator implements the Python iterator protocol. When the `next` function is called (when GDB prints each frame), the iterator checks if this frame decorator, `frame`, is wrapping an inlined frame. If it is not, it returns the existing frame decorator untouched. If it is wrapping an inlined frame, it assumes that the inlined frame was contained within the next oldest frame, `eliding_frame`, which it fetches. It then creates and returns a frame decorator, `ElidingFrameDecorator`, which contains both the elided frame, and the eliding frame.

```
class ElidingInlineDecorator(FrameDecorator):

    def __init__(self, frame, elided_frames):
        super(ElidingInlineDecorator, self).__init__(frame)
        self.frame = frame
        self.elided_frames = elided_frames
```

that an earlier unwinder could not unwind—the value will be unavailable. Instead, the `gdb.Value` returned from this method will be lazy; that is, its underlying bits will not be fetched until it is first used. So, attempting to use such a value will cause an exception at the point of use.

The type of the returned `gdb.Value` depends on the register and the architecture. It is common for registers to have a scalar type, like `long long`; but many other types are possible, such as pointer, pointer-to-function, floating point or vector types.

It also provides a factory method to create a `gdb.UnwindInfo` instance to be returned to GDB:

`PendingFrame.create_unwind_info` (*frame_id*) [Function]

Returns a new `gdb.UnwindInfo` instance identified by given *frame_id*. The argument is used to build GDB's frame ID using one of functions provided by GDB. *frame_id*'s attributes determine which function will be used, as follows:

`sp, pc` The frame is identified by the given stack address and PC. The stack address must be chosen so that it is constant throughout the lifetime of the frame, so a typical choice is the value of the stack pointer at the start of the function—in the DWARF standard, this would be the "Call Frame Address".

This is the most common case by far. The other cases are documented for completeness but are only useful in specialized situations.

`sp, pc, special`
The frame is identified by the stack address, the PC, and a "special" address. The special address is used on architectures that can have frames that do not change the stack, but which are still distinct, for example the IA-64, which has a second stack for registers. Both *sp* and *special* must be constant throughout the lifetime of the frame.

`sp` The frame is identified by the stack address only. Any other stack frame with a matching *sp* will be considered to match this frame. Inside gdb, this is called a "wild frame". You will never need this.

Each attribute value should be an instance of `gdb.Value`.

`PendingFrame.architecture` () [Function]

Return the `gdb.Architecture` (see Section 23.3.2.34 [Architectures In Python], page 460) for this `gdb.PendingFrame`. This represents the architecture of the particular frame being unwound.

`PendingFrame.level` () [Function]

Return an integer, the stack frame level for this frame. See Section 8.1 [Stack Frames], page 107.

Unwinder Output: UnwindInfo

Use `PendingFrame.create_unwind_info` method described above to create a `gdb.UnwindInfo` instance. Use the following method to specify caller registers that have been saved in this frame:

23.3.2.13 Xmethods In Python

Xmethods are additional methods or replacements for existing methods of a C++ class. This feature is useful for those cases where a method defined in C++ source code could be inlined or optimized out by the compiler, making it unavailable to GDB. For such cases, one can define an xmethod to serve as a replacement for the method defined in the C++ source code. GDB will then invoke the xmethod, instead of the C++ method, to evaluate expressions. One can also use xmethods when debugging with core files. Moreover, when debugging live programs, invoking an xmethod need not involve running the inferior (which can potentially perturb its state). Hence, even if the C++ method is available, it is better to use its replacement xmethod if one is defined.

The xmethods feature in Python is available via the concepts of an *xmethod matcher* and an *xmethod worker*. To implement an xmethod, one has to implement a matcher and a corresponding worker for it (more than one worker can be implemented, each catering to a different overloaded instance of the method). Internally, GDB invokes the `match` method of a matcher to match the class type and method name. On a match, the `match` method returns a list of matching *worker* objects. Each worker object typically corresponds to an overloaded instance of the xmethod. They implement a `get_arg_types` method which returns a sequence of types corresponding to the arguments the xmethod requires. GDB uses this sequence of types to perform overload resolution and picks a winning xmethod worker. A winner is also selected from among the methods GDB finds in the C++ source code. Next, the winning xmethod worker and the winning C++ method are compared to select an overall winner. In case of a tie between a xmethod worker and a C++ method, the xmethod worker is selected as the winner. That is, if a winning xmethod worker is found to be equivalent to the winning C++ method, then the xmethod worker is treated as a replacement for the C++ method. GDB uses the overall winner to invoke the method. If the winning xmethod worker is the overall winner, then the corresponding xmethod is invoked via the `__call__` method of the worker object.

If one wants to implement an xmethod as a replacement for an existing C++ method, then they have to implement an equivalent xmethod which has exactly the same name and takes arguments of exactly the same type as the C++ method. If the user wants to invoke the C++ method even though a replacement xmethod is available for that method, then they can disable the xmethod.

See Section 23.3.2.14 [Xmethod API], page 414, for API to implement xmethods in Python. See Section 23.3.2.15 [Writing an Xmethod], page 416, for implementing xmethods in Python.

23.3.2.14 Xmethod API

The GDB Python API provides classes, interfaces and functions to implement, register and manipulate xmethods. See Section 23.3.2.13 [Xmethods In Python], page 414.

An xmethod matcher should be an instance of a class derived from `XMethodMatcher` defined in the module `gdb.xmethod`, or an object with similar interface and attributes. An instance of `XMethodMatcher` has the following attributes:

`name` [Variable]

The name of the matcher.

`enabled` [Variable]
A boolean value indicating whether the matcher is enabled or disabled.

`methods` [Variable]
A list of named methods managed by the matcher. Each object in the list is an instance of the class `XMethod` defined in the module `gdb.xmethod`, or any object with the following attributes:

`name` Name of the xmethod which should be unique for each xmethod managed by the matcher.

`enabled` A boolean value indicating whether the xmethod is enabled or disabled.

The class `XMethod` is a convenience class with same attributes as above along with the following constructor:

`XMethod.__init__` (*self*, *name*) [Function]
Constructs an enabled xmethod with name *name*.

The `XMethodMatcher` class has the following methods:

`XMethodMatcher.__init__` (*self*, *name*) [Function]
Constructs an enabled xmethod matcher with name *name*. The `methods` attribute is initialized to `None`.

`XMethodMatcher.match` (*self*, *class_type*, *method_name*) [Function]
Derived classes should override this method. It should return a xmethod worker object (or a sequence of xmethod worker objects) 9.427 Td .4e177.459 -13class has 1 0 0 1 349.413 404.153 36

A lect wdis oJ/F88 eni-31es n name

8309.003) the match-334(isF56 10.9091 Tf 105.934 70gdb.xmethod)]TJ/F55 10.9091 Tf 63 0 Td [3.199, or disabl

For GDB to lookup xmethods, the xmethod matchers should be registered using the following function defined in the module `gdb.xmethod`:

`register_xmethod_matcher` (*locus, matcher, replace=False*) [Function]
The `matcher` is registered with `locus`, replacing an existing matcher with the same name as `matcher` if `replace` is `True`. `locus` can be a `gdb.Objfile` object (see Section 23.3.2.25 [Objfiles In Python], page 441), or a `gdb.Progspace` object (see Section 23.3.2.24 [Progspaces In Python], page 439), or `None`. If it is `None`, then `matcher` is registered globally.

23.3.2.15 Writing an Xmethod

Implementing xmethods in Python will require implementing xmethod matchers and xmethod workers (see Section 23.3.2.13 [Xmethods In Python], page 414). Consider the following C++ class:

```
class MyClass
{
public:
  MyClass (int a) : a_(a) { }

  int geta (void) { return a_; }
  int operator+ (int b);

private:
  int a_;
};

int
MyClass::operator+ (int b)
{
  return a_ + b;
}
```

Let us define two xmethods for the class `MyClass`, one replacing the method `geta`, and another adding an overloaded flavor of `operator+` which takes a `MyClass` argument (the C++ code above already has an overloaded `operator+` which takes an `int` argument). The xmethod matcher can be defined as follows:

```
class MyClass_geta(gdb.xmethod.XMethod):
    def __init__(self):
        gdb.xmethod.XMethod.__init__(self, 'geta')

    def get_worker(self, method_name):
        if method_name == 'geta':
            return MyClassWorker_geta()


class MyClass_sum(gdb.xmethod.XMethod):
    def __init__(self):
        gdb.xmethod.XMethod.__init__(self, 'sum')

    def get_worker(self, method_name):
        if method_name == 'operator+':
            return MyClassWorker_plus()


class MyClassMatcher(gdb.xmethod.XMethodMatcher):
```

```
    def __init__(self):
        gdb.xmethod.XMethodMatcher.__init__(self, 'MyClassMatcher')
        # List of methods 'managed' by this matcher
        self.methods = [MyClass_geta(), MyClass_sum()]

    def match(self, class_type, method_name):
        if class_type.tag != 'MyClass':
            return None
        workers = []
        for method in self.methods:
            if method.enabled:
                worker = method.get_worker(method_name)
                if worker:
                    workers.append(worker)

        return workers
```

Notice that the `match` method of `MyClassMatcher` returns a worker object of type `MyClassWorker_geta` for the `geta` method, and a worker object of type `MyClassWorker_plus` for the `operator+` method. This is done indirectly via helper classes derived from `gdb.xmethod.XMethod`. One does not need to use the `methods` attribute in a matcher as it is optional. However, if a matcher manages more than one xmethod, it is a good practice to list the xmethods in the `methods` attribute of the matcher. This will then facilitate enabling and disabling individual xmethods via the `enable/disable` commands. Notice also that a worker object is returned only if the corresponding entry in the `methods` attribute of the matcher is enabled.

The implementation of the worker classes returned by the matcher setup above is as follows:

```
class MyClassWorker_geta(gdb.xmethod.XMethodWorker):
    def get_arg_types(self):
        return None

    def get_result_type(self, obj):
        return gdb.lookup_type('int')

    def __call__(self, obj):
        return obj['a_']


class MyClassWorker_plus(gdb.xmethod.XMethodWorker):
    def get_arg_types(self):
        return gdb.lookup_type('MyClass')

    def get_result_type(self, obj):
        return gdb.lookup_type('int')

    def __call__(self, obj, other):
        return obj['a_'] + other['a_']
```

For GDB to actually lookup a xmethod, it has to be registered with it. The matcher defined above is registered with GDB

then, after loading the Python script defining the xmethod matchers and workers into `GDBN`, invoking the method `geta` or using the operator `+` on `obj` will invoke the xmethods defined above:

```
(gdb) p obj.geta()
$1 = 5

(gdb) p obj + obj
$2 = 10
```

Consider another example with a C++ template class:

```
template <class T>
class MyTemplate
{
public:
  MyTemplate () : dsize_(10), data_ (new T [10]) { }
  ~MyTemplate () { delete [] data_; }

  int footprint (void)
  {
    return sizeof (T) * dsize_ + sizeof (MyTemplate<T>);
  }

private:
  int dsize_;
  T *data_;
};
```

Let us implement an xmethod for the above class which serves as a replacement for the `footprint` method. The full code listing of the xmethod workers and xmethod matchers is as follows:

```
class MyTemplateWorker_footprint(gdb.xmethod.XMethodWorker):
    def __init__(self, class_type):
        self.class_type = class_type

    def get_arg_types(self):
        return None

    def get_result_type(self):
        return gdb.lookup_type('int')

    def __call__(self, obj):
        return (self.class_type.sizeof +
                obj['dsize_'] *
                self.class_type.template_argument(0).sizeof)


class MyTemplateMatcher_footprint(gdb.xmethod.XMethodMatcher):
    def __init__(self):
        gdb.xmethod.XMethodMatcher.__init__(self, 'MyTemplateMatcher')

    def match(self, class_type, method_name):
        if (re.match('MyTemplate<[ \t\n]*[_a-zA-Z][ _a-zA-Z0-9]*>',
                     class_type.tag) and
            method_name == 'footprint'):
            return MyTemplateWorker_footprint(class_type)
```

`InferiorCallPreEvent.ptid` [Variable]
The thread in which the call will be run.

`InferiorCallPreEvent.address` [Variable]
The location of the function to be called.

`gdb.InferiorCallPostEvent`
Indicates that a function in the inferior has just been called.

`InferiorCallPostEvent.ptid` [Variable]
The thread in which the call was run.

`InferiorCallPostEvent.address` [Variable]
The location of the function that was called.

`events.memory_changed`
Emits `gdb.MemoryChangedEvent` which indicates that the memory of the inferior has been modified by the GDB user, for instance via a command like `set *addr = value`. The event has the following attributes:

`MemoryChangedEvent.address` [Variable]
The start address of the changed region.

`MemoryChangedEvent.length` [Variable]
Length in bytes of the changed region.

`events.register_changed`
Emits `gdb.RegisterChangedEvent` which indicates that a register in the inferior has been modified by the GDB user.

`RegisterChangedEvent.frame` [Variable]
A gdb.Frame object representing the frame in which the register was modified.

`RegisterChangedEvent.regnum` [Variable]
Denotes which register was modified.

`events.breakpoint_created`
This is emitted when a new breakpoint has been created. The argument that is passed is the new `gdb.Breakpoint` object.

`events.breakpoint_modified`
This is emitted when a breakpoint has been modified in some way. The argument that is passed is the new `gdb.Breakpoint` object.

`events.breakpoint_deleted`
This is emitted when a breakpoint has been deleted. The argument that is passed is the `gdb.Breakpoint` object. When this event is emitted, the `gdb.Breakpoint` object will already be in its invalid state; that is, the `is_valid` method will return `False`.

`events.before_prompt`
: This event carries no payload. It is emitted each time GDB presents a prompt to the user.

`events.new_inferior`
: This is emitted when a new inferior is created. Note that the inferior is not necessarily running; in fact, it may not even have an associated executable.

  The event is of type `gdb.NewInferiorEvent`. This has a single attribute:

  `NewInferiorEvent.inferior` [Variable]
  : The new inferior, a `gdb.Inferior` object.

`events.inferior_deleted`
: This is emitted when an inferior has been deleted. Note that this is not the same as process exit; it is notified when the inferior itself is removed, say via `remove-inferiors`.

  The event is of type `gdb.InferiorDeletedEvent`. This has a single attribute:

  `NewInferiorEvent.inferior` [Variable]
  : The inferior that is being removed, a `gdb.Inferior` object.

`events.new_thread`
: This is emitted when GDB notices a new thread. The event is of type `gdb.NewThreadEvent`, which extends `gdb.ThreadEvent`. This has a single attribute:

  `NewThreadEvent.inferior_thread` [Variable]
  : The new thread.

`events.gdb_exiting`
: This is emitted when GDB exits. This event is not emitted if GDB exits as a result of an internal error, or after an unexpected signal. The event is of type `gdb.GdbExitingEvent`, which has a single attribute:

  `GdbExitingEvent.exit_code` [Variable]
  : An integer, the value of the exit code GDB will return.

`events.connection_removed`
: This is emitted when GDB removes a connection (see Section 23.3.2.36 [Connections In Python], page 462). The event is of type `gdb.ConnectionEvent`. This has a single read-only attribute:

  `ConnectionEvent.connection` [Variable]
  : The `gdb.TargetConnection` that is being removed.

23.3.2.18 Threads In Python

Python scripts can access information about, and manipulate inferior threads controlled by GDB, via objects of the `gdb.InferiorThread` class.

The following thread-related functions are available in the `gdb` module:

`RecordGap.number` [Variable]
: An integer identifying this gap. `number` corresponds to the numbers seen in `record instruction-history` (see Chapter 7 [Process Record and Replay], page 99).

`RecordGap.error_code` [Variable]
: A numerical representation of the reason for the gap. The value is specific to the current recording method.

`RecordGap.error_string` [Variable]
: A human readable string with the reason for the gap.

A `gdb.RecordFunctionSegment` object has the following attributes:

`RecordFunctionSegment.number` [Variable]
: An integer identifying this function segment. `number` corresponds to the numbers seen in `record function-call-history` (see Chapter 7 [Process Record and Replay], page 99).

`RecordFunctionSegment.symbol` [Variable]
: A `gdb.Symbol` object representing the associated symbol. May be `None` if no debug information is available.

`RecordFunctionSegment.level` [Variable]
: An integer representing the function call's stack level. May be `None` if the function call is a gap.

`RecordFunctionSegment.instructions` [Variable]
: A list of `gdb.RecordInstruction` or `gdb.RecordGap` objects associated with this function call.

`RecordFunctionSegment.up` [Variable]
: A `gdb.RecordFunctionSegment` object representing the caller's function segment. If the call has not been recorded, this will be the function segment to which control returns. If neither the call nor the return have been recorded, this will be `None`.

`RecordFunctionSegment.prev` [Variable]
: A `gdb.RecordFunctionSegment` object representing the previous segment of this function call. May be `None`.

`RecordFunctionSegment.next` [Variable]
: A `gdb.RecordFunctionSegment` object representing the next segment of this function call. May be `None`.

The following example demonstrates the usage of these objects and functions to create a function that will rewind a record to the last time a function in a different file was executed. This would typically be used to track the execution of user provided callback functions in a library which typically are not visible in a back trace.

```
def bringback ():
    rec = gdb.current_recording ()
    if not rec:
        return
```

`Command.__init__` (*name*, *command_class* [, *completer_class* [, *prefix*]]) [Function]

The object initializer for `Command` registers the new command with GDB. This initializer is normally invoked from the subclass' own `__init__` method.

name is the name of the command. If *name* consists of multiple words, then the initial words are looked for as prefix commands. In this case, if one of the prefix commands does not exist, an exception is raised.

There is no support for multi-line commands.

command_class should be one of the '`COMMAND_`' constants defined below. This argument tells GDB how to categorize the new command in the help system.

completer_class is an optional argument. If given, it should be one of the '`COMPLETE_`' constants defined below. This argument tells GDB how to perform completion for this command. If not given, GDB will attempt to complete using the object's `complete` method (see below); if no such method is found, an error will occur when completion is attempted.

prefix is an optional argument. If `True`, then the new command is a prefix command; sub-commands of this command may be registered.

The help text for the new command is taken from the Python documentation string for the command's class, if there is one. If no documentation string is provided, the default value "This command is not documented." is used.

`Command.dont_repeat` () [Function]

By default, a GDB command is repeated when the user enters a blank line at the command prompt. A command can suppress this behavior by invoking the `dont_repeat` method. This is similar to the user command `dont-repeat`, see Section 23.1.1 [Define], page 367.

`Command.invoke` (*argument*, *from_tty*) [Function]

This method is called by GDB when this command is invoked.

argument is a string. It is the argument to the command, after leading and trailing whitespace has been stripped.

from_tty is a boolean argument. When true, this means that the command was entered by the user at the terminal; when false it means that the command came from elsewhere.

If this method throws an exception, it is turned into a GDB `error` call. Otherwise, the return value is ignored.

To break *argument* up into an argv-like string use `gdb.string_to_argv`. This function behaves identically to GDB's internal argument lexer `buildargv`. It is recommended to use this for consistency. Arguments are separated by spaces and may be quoted. Example:

```
print gdb.string_to_argv ("1 2\ \\\"3 '4 \"5' \"6 '7\"")
['1', '2 "3', '4 "5', "6 '7"]
```

`Command.complete` (*text*, *word*) [Function]

This method is called by GDB when the user attempts completion on this command. All forms of completion are handled by this method, that is, the `TAB` and `M-?` key

bindings (see Section 3.3 [Completion], page 24), and the `complete` command (see Section 3.5 [Help], page 28).

The arguments *text* and *word* are both strings; *text* holds the complete command line up to the cursor's location, while *word* holds the last word of the command line; this is computed using a word-breaking heuristic.

The `complete` method can return several values:

- If the return value is a sequence, the contents of the sequence are used as the completions. It is up to `complete` to ensure that the contents actually do complete the word. A zero-length sequence is allowed, it means that there were no completions available. Only string elements of the sequence are used; other elements in the sequence are ignored.
- If the return value is one of the '`COMPLETE_`' constants defined below, then the corresponding GDB-internal completion function is invoked, and its result is used.
- All other results are treated as though there were no available completions.

When a new command is registered, it must be declared as a member of some general class of commands. This is used to classify top-level commands in the on-line help system; note that prefix commands are not listed under their own category but rather that of their top-level command. The available classifications are represented by constants defined in the `gdb` module:

`gdb.COMMAND_NONE`
: The command does not belong to any particular class. A command in this category will not be displayed in any of the help categories.

`gdb.COMMAND_RUNNING`
: The command is related to running the inferior. For example, `start`, `step`, and `continue` are in this category. Type ***help running*** at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_DATA`
: The command is related to data or variables. For example, `call`, `find`, and `print` are in this category. Type ***help data*** at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_STACK`
: The command has to do with manipulation of the stack. For example, `backtrace`, `frame`, and `return` are in this category. Type ***help stack*** at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_FILES`
: This class is used for file-related commands. For example, `file`, `list` and `section` are in this category. Type ***help files*** at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_SUPPORT`
: This should be used for "support facilities", generally meaning things that are useful to the user when interacting with GDB, but not related to the state of the inferior. For example, `help`, `make`, and `shell` are in this category. Type ***help support*** at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_STATUS`
: The command is an '`info`'-related command, that is, related to the state of GDB itself. For example, `info`, `macro`, and `show` are in this category. Type *`help status`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_BREAKPOINTS`
: The command has to do with breakpoints. For example, `break`, `clear`, and `delete` are in this category. Type *`help breakpoints`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_TRACEPOINTS`
: The command has to do with tracepoints. For example, `trace`, `actions`, and `tfind` are in this category. Type *`help tracepoints`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_TUI`
: The command has to do with the text user interface (see Chapter 25 [TUI], page 527). Type *`help tui`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_USER`
: The command is a general purpose command for the user, and typically does not fit in one of the other categories. Type *`help user-defined`* at the GDB prompt to see a list of commands in this category, as well as the list of gdb macros (see Section 23.1 [Sequences], page 367).

`gdb.COMMAND_OBSCURE`
: The command is only used in unusual circumstances, or is not of general interest to users. For example, `checkpoint`, `fork`, and `stop` are in this category. Type *`help obscure`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_MAINTENANCE`
: The command is only useful to GDB maintainers. The `maintenance` and `flushregs` commands are in this category. Type *`help internals`* at the GDB prompt to see a list of commands in this category.

A new command can use a predefined completion function, either by specifying it via an argument at initialization, or by returning it from the `complete` method. These predefined completion constants are all defined in the `gdb` module:

`gdb.COMPLETE_NONE`
: This constant means that no completion should be done.

`gdb.COMPLETE_FILENAME`
: This constant means that filename completion should be performed.

`gdb.COMPLETE_LOCATION`
: This constant means that location completion should be done. See Section 9.2 [Specify Location], page 120.

`gdb.COMPLETE_COMMAND`
: This constant means that completion should examine GDB command names.

gdb.COMPLETE_SYMBOL
: This constant means that completion should be done using symbol names as the source.

gdb.COMPLETE_EXPRESSION
: This constant means that completion should be done on expressions. Often this means completing on symbol names, but some language parsers also have support for completing on fields names.

The following code snippet shows how a trivial CLI command can be implemented in Python:

```
class HelloWorld (gdb.Command):
  """Greet the whole world."""

  def __init__ (self):
    super (HelloWorld, self).__init__ ("hello-world", gdb.COMMAND_USER)

  def invoke (self, arg, from_tty):
    print ("Hello, World!")

HelloWorld ()
```

The last line instantiates the class, and is necessary to trigger the registration of the command with gdb. Depending on how the Python code is read into gdb, you may need to import the gdb module explicitly.

23.3.2.21 GDB/MI Commands In Python

It is possible to add GDB/MI (see Chapter 27 [GDB/MI], page 537) commands implemented in Python. A GDB/MI command is implemented using an instance of the gdb.MICommand class, most commonly using a subclass.

MICommand.__init__ (name) [Function]
: The object initializer for MICommand registers the new command with gdb. This initializer is normally invoked from the subclass' own __init__ method.

  name is the name of the command. It must be a valid name of a GDB/MI command, and in particular must start with a hyphen (-). Reusing the name of a built-in GDB/MI is not allowed, and a RuntimeError will be raised. Using the name of an GDB/MI command previously defined in Python is allowed, the previous command will be replaced with the new command.

MICommand.invoke(arguments) [Function]
: This method is called by gdb when the new MI command is invoked.

  arguments is a list of strings. Note, that --thread and --frame arguments are handled by gdb itself therefore they do not show up in arguments.

  If this method raises an exception, then it is turned into a GDB/MI ^error response. Only gdb.GdbError exceptions (or its sub-classes) should be used for reporting errors to users, any other exception type is treated as a failure of the invoke method, and the exception will be printed to the error stream according to the set python print-stack setting (see [set python print-stack], page 377).

  If this method returns None, then the GDB/MI command will return a ^done response with no additional values.

`data-directory` [Scheme Procedure]
Return a string containing GDB's data directory. This directory contains GDB's ancillary files.

`guile-data-directory` [Scheme Procedure]
Return a string containing GDB's Guile data directory. This directory contains the Guile modules provided by GDB.

`gdb-version` [Scheme Procedure]
Return a string containing the GDB version.

`host-config` [Scheme Procedure]
Return a string containing the host configuration. This is the string passed to `--host` when GDB was configured.

`target-config` [Scheme Procedure]
Return a string containing the target configuration. This is the string passed to `--target` when GDB was configured.

23.4.3.3 GDB Scheme Data Types

The values exposed by GDB to Guile are known as GDB *objects*. There are several kinds of GDB object, and each is disjoint from all other types known to Guile.

`gdb-object-kind` *object* [Scheme Procedure]
Return the kind of the GDB object, e.g., `<gdb:breakpoint>`, as a symbol.

GDB defines the following object types:

`<gdb:arch>`
See Section 23.4.3.21 [Architectures In Guile], page 513.

`<gdb:block>`
See Section 23.4.3.16 [Blocks In Guile], page 502.

`<gdb:block-symbols-iterator>`
See Section 23.4.3.16 [Blocks In Guile], page 502.

`<gdb:breakpoint>`
See Section 23.4.3.19 [Breakpoints In Guile], page 508.

`<gdb:command>`
See Section 23.4.3.11 [Commands In Guile], page 491.

`<gdb:exception>`
See Section 23.4.3.4 [Guile Exception Handling], page 473.

`<gdb:frame>`
See Section 23.4.3.15 [Frames In Guile], page 499.

`<gdb:iterator>`
See Section 23.4.3.25 [Iterators In Guile], page 517.

`<gdb:lazy-string>`
See Section 23.4.3.20 [Lazy Strings In Guile], page 512.

`<gdb:objfile>`
: See Section 23.4.3.14 [Objfiles In Guile], page 498.

`<gdb:parameter>`
: See Section 23.4.3.12 [Parameters In Guile], page 495.

`<gdb:pretty-printer>`
: See Section 23.4.3.8 [Guile Pretty Printing API], page 486.

`<gdb:pretty-printer-worker>`
: See Section 23.4.3.8 [Guile Pretty Printing API], page 486.

`<gdb:progspace>`
: See Section 23.4.3.13 [Progspaces In Guile], page 497.

`<gdb:symbol>`
: See Section 23.4.3.17 [Symbols In Guile], page 503.

`<gdb:symtab>`
: See Section 23.4.3.18 [Symbol Tables In Guile], page 507.

`<gdb:sal>`
: See Section 23.4.3.18 [Symbol Tables In Guile], page 507.

`<gdb:type>`
: See Section 23.4.3.7 [Types In Guile], page 482.

`<gdb:field>`
: See Section 23.4.3.7 [Types In Guile], page 482.

`<gdb:value>`
: See Section 23.4.3.5 [Values From Inferior In Guile], page 475.

The following GDB objects are managed internally so that the Scheme function `eq?` may be applied to them.

```
<gdb:arch>
<gdb:block>
<gdb:breakpoint>
<gdb:frame>
<gdb:objfile>
<gdb:progspace>
<gdb:symbol>
<gdb:symtab>
<gdb:type>
```

23.4.3.4 Guile Exception Handling

When executing the `guile` command, Guile exceptions uncaught within the Guile code are translated to calls to the GDB error-reporting mechanism. If the command that called `guile` does not handle the error, GDB will terminate it and report the error according to the setting of the `guile print-stack` parameter.

The `guile print-stack` parameter has three settings:

`none`
: Nothing is printed.

`gdb:memory-error`
: This exception is thrown when an operation tried to access invalid memory in the inferior.

`gdb:pp-type-error`
: This exception is thrown when a Guile pretty-printer passes a bad object to GDB.

The following exception-related procedures are provided by the `(gdb)` module.

`make-exception` *key args* [Scheme Procedure]
: Return a `<gdb:exception>` object given by its *key* and *args*, which are the standard Guile parameters of an exception. See the Guile documentation for more information (see Section "Exceptions" in *GNU Guile Reference Manual*).

`exception?` *object* [Scheme Procedure]
: Return `#t` if *object* is a `<gdb:exception>` object. Otherwise return `#f`.

`exception-key` *exception* [Scheme Procedure]
: Return the *args* field of a `<gdb:exception>` object.

`exception-args` *exception* [Scheme Procedure]
: Return the *args* field of a `<gdb:exception>` object.

23.4.3.5 Values From Inferior In Guile

GDB provides values it obtains from the inferior program in an object of type `<gdb:value>`. GDB uses this object for its internal bookkeeping of the inferior's values, and for fetching values when necessary.

GDB does not memoize `<gdb:value>` objects. `make-value` always returns a fresh object.

```
(gdb) guile (eq? (make-value 1) (make-value 1))
$1 = #f
(gdb) guile (equal? (make-value 1) (make-value 1))
$1 = #t
```

A `<gdb:value>` that represents a function can be executed via inferior function call with `value-call`. Any arguments provided to the call must match the function's prototype, and must be provided in the order specified by that prototype.

For example, `some-val` is a `<gdb:value>` instance representing a function that takes two integers as arguments. To execute this function, call it like so:

```
(define result (value-call some-val 10 20))
```

Any values returned from a function call are `<gdb:value>` objects.

Note: Unlike Python scripting in GDB, inferior values that are simple scalars cannot be used directly in Scheme expressions that are valid for the value's data type. For example, `(+ (parse-and-eval "int_variable") 2)` does not work. And inferior values that are structures or instances of some class cannot be accessed using any special syntax, instead `value-field` must be used.

The following value-related procedures are provided by the `(gdb)` module.

`value?` *object* [Scheme Procedure]
: Return `#t` if *object* is a `<gdb:value>` object. Otherwise return `#f`.

`value-address` *value* [Scheme Procedure]
If *value* is addressable, returns a `<gdb:value>` object representing the address. Otherwise, `#f` is returned.

`value-type` *value* [Scheme Procedure]
Return the type of *value* as a `<gdb:type>` object (see Section 23.4.3.7 [Types In Guile], page 482).

`value-dynamic-type` *value* [Scheme Procedure]
Return the dynamic type of *value*. This uses C++ run-time type information (RTTI) to determine the dynamic type of the value. If the value is of class type, it will return the class in which the value is embedded, if any. If the value is of pointer or reference to a class type, it will compute the dynamic type of the referenced object, and return a pointer or reference to that type, respectively. In all other cases, it will return the value's static type.

Note that this feature will only work when debugging a C++ program that includes RTTI for the object in question. Otherwise, it will just return the static type of the value as in *`ptype foo`*. See Chapter 16 [Symbols], page 251.

`value-cast` *value type* [Scheme Procedure]
Return a new instance of `<gdb:value>` that is the result of casting *value* to the type described by *type*, which must be a `<gdb:type>` object. If the cast cannot be performed for some reason, this method throws an exception.

`value-dynamic-cast` *value type* [Scheme Procedure]
Like `value-cast`, but works as if the C++ `dynamic_cast` operator were used. Consult a C++ reference for details.

`value-reinterpret-cast` *value type* [Scheme Procedure]
Like `value-cast`, but works as if the C++ `reinterpret_cast` operator were used. Consult a C++ reference for details.

`value-dereference` *value* [Scheme Procedure]
For pointer data types, this method returns a new `<gdb:value>` object whose contents is the object pointed to by *value*. For example, if `foo` is a C pointer to an `int`, declared in your C program as

```
int *foo;
```

then you can use the corresponding `<gdb:value>` to access what `foo` points to like this:

```
(define bar (value-dereference foo))
```

The result `bar` will be a `<gdb:value>` object holding the value pointed to by `foo`.

A similar function `value-referenced-value` exists which also returns `<gdb:value>` objects corresponding to the values pointed to by pointer values (and additionally, values referenced by reference values). However, the behavior of `value-dereference` differs from `value-referenced-value` by the fact that the behavior of `value-dereference` is identical to applying the C unary operator `*` on a given value. For example, consider a reference to a pointer `ptrref`, declared in your C++ program as

```
typedef int *intptr;
```

```
...
int val = 10;
intptr ptr = &val;
intptr &ptrref = ptr;
```

Though `ptrref` is a reference value, one can apply the method `value-dereference` to the `<gdb:value>` object corresponding to it and obtain a `<gdb:value>` which is identical to that corresponding to `val`. However, if you apply the method `value-referenced-value`, the result would be a `<gdb:value>` object identical to that corresponding to `ptr`.

```
(define scm-ptrref (parse-and-eval "ptrref"))
(define scm-val (value-dereference scm-ptrref))
(define scm-ptr (value-referenced-value scm-ptrref))
```

The `<gdb:value>` object `scm-val` is identical to that corresponding to `val`, and `scm-ptr` is identical to that corresponding to `ptr`. In general, `value-dereference` can be applied whenever the C unary operator `*` can be applied to the corresponding C value. For those cases where applying both `value-dereference` and `value-referenced-value` is allowed, the results obtained need not be identical (as we have seen in the above example). The results are however identical when applied on `<gdb:value>` objects corresponding to pointers (`<gdb:value>` objects with type code `TYPE_CODE_PTR`) in a C/C++ program.

`value-referenced-value` *value* [Scheme Procedure]

For pointer or reference data types, this method returns a new `<gdb:value>` object corresponding to the value referenced by the pointer/reference value. For pointer data types, `value-dereference` and `value-referenced-value` produce identical results. The difference between these methods is that `value-dereference` cannot get the values referenced by reference values. For example, consider a reference to an `int`, declared in your C++ program as

```
int val = 10;
int &ref = val;
```

then applying `value-dereference` to the `<gdb:value>` object corresponding to `ref` will result in an error, while applying `value-referenced-value` will result in a `<gdb:value>` object identical to that corresponding to `val`.

```
(define scm-ref (parse-and-eval "ref"))
(define err-ref (value-dereference scm-ref))      ;; error
(define scm-val (value-referenced-value scm-ref)) ;; ok
```

The `<gdb:value>` object `scm-val` is identical to that corresponding to `val`.

`value-reference-value` *value* [Scheme Procedure]

Return a new `<gdb:value>` object which is a reference to the value encapsulated by `<gdb:value>` object *value*.

`value-rvalue-reference-value` *value* [Scheme Procedure]

Return a new `<gdb:value>` object which is an rvalue reference to the value encapsulated by `<gdb:value>` object *value*.

`value-const-value` *value* [Scheme Procedure]

Return a new `<gdb:value>` object which is a '`const`' version of `<gdb:value>` object *value*.

`value-field` *value field-name* [Scheme Procedure]
Return field *field-name* from `<gdb:value>` object *value*.

`value-subscript` *value index* [Scheme Procedure]
Return the value of array *value* at index *index*. The *value* argument must be a subscriptable `<gdb:value>` object.

`value-call` *value arg-list* [Scheme Procedure]
Perform an inferior function call, taking *value* as a pointer to the function to call. Each element of list *arg-list* must be a <gdb:value> object or an object that can be converted to a value. The result is the value returned by the function.

`value->bool` *value* [Scheme Procedure]
Return the Scheme boolean representing `<gdb:value>` *value*. The value must be "integer like". Pointers are ok.

`value->integer` [Scheme Procedure]
Return the Scheme integer representing `<gdb:value>` *value*. The value must be "integer like". Pointers are ok.

`value->real` [Scheme Procedure]
Return the Scheme real number representing `<gdb:value>` *value*. The value must be a number.

`value->bytevector` [Scheme Procedure]
Return a Scheme bytevector with the raw contents of `<gdb:value>` *value*. No transformation, endian or otherwise, is performed.

`value->string` *value* [*#:encoding encoding*] [Scheme Procedure]
[*#:errors errors*] [*#:length length*]
If *value*> represents a string, then this method converts the contents to a Guile string. Otherwise, this method will throw an exception.

Values are interpreted as strings according to the rules of the current language. If the optional length argument is given, the string will be converted to that length, and will include any embedded zeroes that the string may contain. Otherwise, for languages where the string is zero-terminated, the entire string will be converted.

For example, in C-like languages, a value is a string if it is a pointer to or an array of characters or ints of type `wchar_t`, `char16_t`, or `char32_t`.

If the optional *encoding* argument is given, it must be a string naming the encoding of the string in the `<gdb:value>`, such as `"ascii"`, `"iso-8859-6"` or `"utf-8"`. It accepts the same encodings as the corresponding argument to Guile's `scm_from_stringn` function, and the Guile codec machinery will be used to convert the string. If *encoding* is not given, or if *encoding* is the empty string, then either the `target-charset` (see Section 10.21 [Character Sets], page 177) will be used, or a language-specific encoding will be used, if the current language is able to supply one.

The optional *errors* argument is one of `#f`, `error` or `substitute`. `error` and `substitute` must be symbols. If *errors* is not specified, or if its value is `#f`, then the default conversion strategy is used, which is set with the Scheme function

`set-port-conversion-strategy!`. If the value is `'error` then an exception is thrown if there is any conversion error. If the value is `'substitute` then any conversion error is replaced with question marks. See Section "Strings" in *GNU Guile Reference Manual*.

If the optional *length* argument is given, the string will be fetched and converted to the given length. The length must be a Scheme integer and not a `<gdb:value>` integer.

`value->lazy-string` *value* [*#:encoding encoding*] [*#:length length*] [Scheme Procedure]

If this `<gdb:value>` represents a string, then this method converts *value* to a `<gdb:lazy-string` (see Section 23.4.3.20 [Lazy Strings In Guile], page 512). Otherwise, this method will throw an exception.

If the optional *encoding* argument is given, it must be a string naming the encoding of the `<gdb:lazy-string`. Some examples are: `"ascii"`, `"iso-8859-6"` or `"utf-8"`. If the *encoding* argument is an encoding that GDB does not recognize, GDB will raise an error.

When a lazy string is printed, the GDB encoding machinery is used to convert the string during printing. If the optional *encoding* argument is not provided, or is an empty string, GDB will automatically select the encoding most suitable for the string type. For further information on encoding in GDB please see Section 10.21 [Character Sets], page 177.

If the optional *length* argument is given, the string will be fetched and encoded to the length of characters specified. If the *length* argument is not provided, the string will be fetched and encoded until a null of appropriate width is found. The length must be a Scheme integer and not a `<gdb:value>` integer.

`value-lazy?` *value* [Scheme Procedure]

Return `#t` if *value* has not yet been fetched from the inferior. Otherwise return `#f`. GDB does not fetch values until necessary, for efficiency. For example:

```
(define myval (parse-and-eval "somevar"))
```

The value of `somevar` is not fetched at this time. It will be fetched when the value is needed, or when the `fetch-lazy` procedure is invoked.

`make-lazy-value` *type address* [Scheme Procedure]

Return a `<gdb:value>` that will be lazily fetched from the target. The object of type `<gdb:type>` whose value to fetch is specified by its *type* and its target memory *address*, which is a Scheme integer.

`value-fetch-lazy!` *value* [Scheme Procedure]

If *value* is a lazy value (`(value-lazy? value)` is `#t`), then the value is fetched from the inferior. Any errors that occur in the process will produce a Guile exception.

If *value* is not a lazy value, this method has no effect.

The result of this function is unspecified.

`value-print` *value* [Scheme Procedure]

Return the string representation (print form) of `<gdb:value>` *value*.

23.4.3.6 Arithmetic In Guile

The `(gdb)` module provides several functions for performing arithmetic on `<gdb:value>` objects. The arithmetic is performed as if it were done by the target, and therefore has target semantics which are not necessarily those of Scheme. For example operations work with a fixed precision, not the arbitrary precision of Scheme.

Wherever a function takes an integer or pointer as an operand, GDB will convert appropriate Scheme values to perform the operation.

`value-add` *a b* [Scheme Procedure]

`value-sub` *a b* [Scheme Procedure]

`value-mul` *a b* [Scheme Procedure]

`value-div` *a b* [Scheme Procedure]

`value-rem` *a b* [Scheme Procedure]

`value-mod` *a b* [Scheme Procedure]

`value-pow` *a b* [Scheme Procedure]

`value-not` *a* [Scheme Procedure]

`value-neg` *a* [Scheme Procedure]

`value-pos` *a* [Scheme Procedure]

`value-abs` *a* [Scheme Procedure]

`value-lsh` *a b* [Scheme Procedure]

`value-rsh` *a b* [Scheme Procedure]

`value-min` *a b* [Scheme Procedure]

`value-max` *a b* [Scheme Procedure]

`value-lognot` *a* [Scheme Procedure]

`value-logand` *a b* [Scheme Procedure]

`value-logior` *a b* [Scheme Procedure]

`value-logxor` *a b* [Scheme Procedure]

`value=?` *a b* [Scheme Procedure]

`value<?` *a b* [Scheme Procedure]

`value<=?` *a b* [Scheme Procedure]

`value>?` *a b* [Scheme Procedure]

`value>=?` *a b* [Scheme Procedure]

Scheme does not provide a `not-equal` function, and thus Guile support in GDB does not either.

23.4.3.7 Types In Guile

GDB represents types from the inferior in objects of type `<gdb:type>`.

The following type-related procedures are provided by the `(gdb)` module.

`type?` *object* [Scheme Procedure]
Return `#t` if *object* is an object of type `<gdb:type>`. Otherwise return `#f`.

`lookup-type` *name* [*#:block block*] [Scheme Procedure]
This function looks up a type by its *name*, which must be a string.

If *block* is given, it is an object of type `<gdb:block>`, and *name* is looked up in that scope. Otherwise, it is searched for globally.

Ordinarily, this function will return an instance of `<gdb:type>`. If the named type cannot be found, it will throw an exception.

`type-code` *type* [Scheme Procedure]
Return the type code of *type*. The type code will be one of the `TYPE_CODE_` constants defined below.

`type-tag` *type* [Scheme Procedure]
Return the tag name of *type*. The tag name is the name after `struct`, `union`, or `enum` in C and C++; not all languages have this concept. If this type has no tag name, then `#f` is returned.

`type-name` *type* [Scheme Procedure]
Return the name of *type*. If this type has no name, then `#f` is returned.

`type-print-name` *type* [Scheme Procedure]
Return the print name of *type*. This returns something even for anonymous types. For example, for an anonymous C struct `"struct {...}"` is returned.

`type-sizeof` *type* [Scheme Procedure]
Return the size of this type, in target `char` units. Usually, a target's `char` type will be an 8-bit byte. However, on some unusual platforms, this type may have a different size.

`type-strip-typedefs` *type* [Scheme Procedure]
Return a new `<gdb:type>` that represents the real type of *type*, after removing all layers of typedefs.

`type-array` *type n1* [*n2*] [Scheme Procedure]
Return a new `<gdb:type>` object which represents an array of this type. If one argument is given, it is the inclusive upper bound of the array; in this case the lower bound is zero. If two arguments are given, the first argument is the lower bound of the array, and the second argument is the upper bound of the array. An array's length must not be negative, but the bounds can be.

`type-vector` *type n1* [*n2*] [Scheme Procedure]
Return a new `<gdb:type>` object which represents a vector of this type. If one argument is given, it is the inclusive upper bound of the vector; in this case the lower

bound is zero. If two arguments are given, the first argument is the lower bound of the vector, and the second argument is the upper bound of the vector. A vector's length must not be negative, but the bounds can be.

The difference between an `array` and a `vector` is that arrays behave like in C: when used in expressions they decay to a pointer to the first element whereas vectors are treated as first class values.

`type-pointer` *type* [Scheme Procedure]
Return a new `<gdb:type>` object which represents a pointer to *type*.

`type-range` *type* [Scheme Procedure]
Return a list of two elements: the low bound and high bound of *type*. If *type* does not have a range, an exception is thrown.

`type-reference` *type* [Scheme Procedure]
Return a new `<gdb:type>` object which represents a reference to *type*.

`type-target` *type* [Scheme Procedure]
Return a new `<gdb:type>` object which represents the target type of *type*.

For a pointer type, the target type is the type of the pointed-to object. For an array type (meaning C-like arrays), the target type is the type of the elements of the array. For a function or method type, the target type is the type of the return value. For a complex type, the target type is the type of the elements. For a typedef, the target type is the aliased type.

If the type does not have a target, this method will throw an exception.

`type-const` *type* [Scheme Procedure]
Return a new `<gdb:type>` object which represents a `const`-qualified variant of *type*.

`type-volatile` *type* [Scheme Procedure]
Return a new `<gdb:type>` object which represents a `volatile`-qualified variant of *type*.

`type-unqualified` *type* [Scheme Procedure]
Return a new `<gdb:type>` object which represents an unqualified variant of *type*. That is, the result is neither `const` nor `volatile`.

`type-num-fields` [Scheme Procedure]
Return the number of fields of `<gdb:type>` *type*.

`type-fields` *type* [Scheme Procedure]
Return the fields of *type* as a list. For structure and union types, `fields` has the usual meaning. Range types have two fields, the minimum and maximum values. Enum types have one field per enum constant. Function and method types have one field per parameter. The base types of C++ classes are also represented as fields. If the type has no fields, or does not fit into one of these categories, an empty list will be returned. See [Fields of a type in Guile], page 485.

`make-field-iterator` *type* [Scheme Procedure]
Return the fields of *type* as a <gdb:iterator> object. See Section 23.4.3.25 [Iterators In Guile], page 517.

`type-field` *type field-name* [Scheme Procedure]
Return field named *field-name* in *type*. The result is an object of type `<gdb:field>`. See [Fields of a type in Guile], page 485. If the type does not have fields, or *field-name* is not a field of *type*, an exception is thrown.

For example, if `some-type` is a `<gdb:type>` instance holding a structure type, you can access its `foo` field with:

```
(define bar (type-field some-type "foo"))
```

`bar` will be a `<gdb:field>` object.

`type-has-field?` *type name* [Scheme Procedure]
Return `#t` if `<gdb:type>` *type* has field named *name*. Otherwise return `#f`.

Each type has a code, which indicates what category this type falls into. The available type categories are represented by constants defined in the `(gdb)` module:

`TYPE_CODE_PTR`
The type is a pointer.

`TYPE_CODE_ARRAY`
The type is an array.

`TYPE_CODE_STRUCT`
The type is a structure.

`TYPE_CODE_UNION`
The type is a union.

`TYPE_CODE_ENUM`
The type is an enum.

`TYPE_CODE_FLAGS`
A bit flags type, used for things such as status registers.

`TYPE_CODE_FUNC`
The type is a function.

`TYPE_CODE_INT`
The type is an integer type.

`TYPE_CODE_FLT`
A floating point type.

`TYPE_CODE_VOID`
The special type `void`.

`TYPE_CODE_SET`
A Pascal set type.

`TYPE_CODE_RANGE`
A range type, that is, an integer type with bounds.

`TYPE_CODE_STRING`
A string type. Note that this is only used for certain languages with language-defined string types; C strings are not represented this way.

`field-type` *field* [Scheme Procedure]
Return the type of the field. This is usually an instance of `<gdb:type>`, but it can be `#f` in some situations.

`field-enumval` *field* [Scheme Procedure]
Return the enum value represented by `<gdb:field>` *field*.

`field-bitpos` *field* [Scheme Procedure]
Return the bit position of `<gdb:field>` *field*. This attribute is not available for `static` fields (as in C++).

`field-bitsize` *field* [Scheme Procedure]
If the field is packed, or is a bitfield, return the size of `<gdb:field>` *field* in bits. Otherwise, zero is returned; in which case the field's size is given by its type.

`field-artificial?` *field* [Scheme Procedure]
Return `#t` if the field is artificial, usually meaning that it was provided by the compiler and not the user. Otherwise return `#f`.

`field-base-class?` *field* [Scheme Procedure]
Return `#t` if the field represents a base class of a C++ structure. Otherwise return `#f`.

23.4.3.8 Guile Pretty Printing API

An example output is provided (see Section 10.10 [Pretty Printing], page 159).

A pretty-printer is represented by an object of type <gdb:pretty-printer>. Pretty-printer objects are created with `make-pretty-printer`.

The following pretty-printer-related procedures are provided by the `(gdb)` module:

`make-pretty-printer` *name lookup-function* [Scheme Procedure]
Return a `<gdb:pretty-printer>` object named *name*.

lookup-function is a function of one parameter: the value to be printed. If the value is handled by this pretty-printer, then *lookup-function* returns an object of type <gdb:pretty-printer-worker> to perform the actual pretty-printing. Otherwise *lookup-function* returns `#f`.

`pretty-printer?` *object* [Scheme Procedure]
Return `#t` if *object* is a `<gdb:pretty-printer>` object. Otherwise return `#f`.

`pretty-printer-enabled?` *pretty-printer* [Scheme Procedure]
Return `#t` if *pretty-printer* is enabled. Otherwise return `#f`.

`set-pretty-printer-enabled!` *pretty-printer flag* [Scheme Procedure]
Set the enabled flag of *pretty-printer* to *flag*. The value returned is unspecified.

`pretty-printers` [Scheme Procedure]
Return the list of global pretty-printers.

`set-pretty-printers!` *pretty-printers* [Scheme Procedure]
Set the list of global pretty-printers to *pretty-printers*. The value returned is unspecified.

make-pretty-printer-worker display-hint to-string children [Scheme Procedure]

Return an object of type <gdb:pretty-printer-worker> .

This function takes three parameters:

`display-hint '

display-hint provides a hint to gdb or gdb front end via MI to change the formatting of the value being printed. The value must be a string or #f (meaning there is no hint). Several values for display-hint are predefined by gdb:

`array ' Indicate that the object being printed is "array-like". The CLI uses this to respect parameters such as set print elements and set print array .

`map' Indicate that the object being printed is "map-like", and that the children of this value can be assumed to alternate between keys and values.

`string ' Indicate that the object being printed is "string-like". If the printer's to-string function returns a Guile string of some kind, then gdb will call its internal language-specific string-printing function to format the string. For the CLI this means adding quotation marks, possibly escaping some characters, respecting set print elements , and the like.

`to-string '

to-string is either a function of one parameter, the <gdb:pretty-printer-worker> object, or #f .

When printing from the CLI, if the to-string method exists, then gdb will prepend its result to the values returned by children . Exactly how this formatting is done is dependent on the display hint, and may change as more hints are added. Also, depending on the print settings (see Section 10.9 [Print Settings], page 149), the CLI may print just the result of to-string in a stack trace, omitting the result of children .

If this method returns a string, it is printed verbatim.

Otherwise, if this method returns an instance of <gdb:value> , then gdb prints this value. This may result in a call to another pretty-printer.

If instead the method returns a Guile value which is convertible to a <gdb:value> , then gdb performs the conversion and prints the resulting value. Again, this may result in a call to another pretty-printer. Guile scalars (integers, floats, and booleans) and strings are convertible to <gdb:value> ; other types are not.

Finally, if this method returns #f then no further operations are peformed in this method and nothing is printed.

If the result is not one of these types, an exception is raised.

to-string may also be #f in which case it is left to children to print the value.

C-c C-s Execute to another source line, like the GDB `step` command; also update the display window to show the current file and location.

C-c C-n Execute to next source line in this function, skipping all function calls, like the GDB `next` command. Then update the display window to show the current file and location.

C-c C-i Execute one instruction, like the GDB `stepi` command; update display window accordingly.

C-c C-f Execute until exit from the selected stack frame, like the GDB `finish` command.

C-c C-r Continue execution of your program, like the GDB `continue` command.

C-c < Go up the number of frames indicated by the numeric argument (see Section "Numeric Arguments" in *The GNU Emacs Manual*), like the GDB `up` command.

C-c > Go down the number of frames indicated by the numeric argument, like the GDB `down` command.

In any source file, the Emacs command *C-x SPC* (`gud-break`) tells GDB to set a breakpoint on the source line point is on.

In text command mode, if you type *M-x speedbar*, Emacs displays a separate frame which shows a backtrace when the GUD buffer is current. Move point to any frame in the stack and type `RET` to make it become the current frame and display the associated source in the source buffer. Alternatively, click *Mouse-2* to make the selected frame become the current one. In graphical mode, the speedbar displays watch expressions.

If you accidentally delete the source-display buffer, an easy way to get it back is to type the command `f` in the GDB buffer, to request a frame display; when you run under Emacs, this recreates the source buffer if necessary to show you the context of the current frame.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of line numbers. If you add or delete lines from the text, the line numbers that GDB knows cease to correspond properly with the code.

A more detailed description of Emacs' interaction with GDB is given in the Emacs manual (see Section "Debuggers" in *The GNU Emacs Manual*).

No frontend is known to do this exactly right, so it is suggested to just always pass the '`--thread`' and '`--frame`' options.

27.3.1.2 Language

The execution of several commands depends on which language is selected. By default, the current language (see [show language], page 219) is used. But for commands known to be language-sensitive, it is recommended to use the '`--language`' option. This option takes one argument, which is the name of the language to use while executing the command. For instance:

```
-data-evaluate-expression --language c "sizeof (void*)"
^done,value="4"
(gdb)
```

The valid language names are the same names accepted by the '`set language`' command (see Section 15.1.2 [Manually], page 218), excluding '`auto`', '`local`' or '`unknown`'.

27.3.2 Asynchronous command execution and non-stop mode

On some targets, GDB is capable of processing MI commands even while the target is running. This is called *asynchronous command execution* (see Section 5.5.3 [Background Execution], page 92). The frontend may specify a preference for asynchronous execution using the `-gdb-set mi-async 1` command, which should be emitted before either running the executable or attaching to the target. After the frontend has started the executable or attached to the target, it can find if asynchronous execution is enabled using the `-list-target-features` command.

`-gdb-set mi-async` [`on|off`]
: Set whether MI is in asynchronous mode.

  When `off`, which is the default, MI execution commands (e.g., `-exec-continue`) are foreground commands, and GDB waits for the program to stop before processing further commands.

  When `on`, MI execution commands are background execution commands (e.g., `-exec-continue` becomes the equivalent of the `c&` CLI command), and so GDB is capable of processing MI commands even while the target is running.

`-gdb-show mi-async`
: Show whether MI asynchronous mode is enabled.

Note: In GDB version 7.7 and earlier, this option was called `target-async` instead of `mi-async`, and it had the effect of both putting MI in asynchronous mode and making CLI background commands possible. CLI background commands are now always possible "out of the box" if the target supports them. The old spelling is kept as a deprecated alias for backwards compatibility.

Even if GDB can accept a command while target is running, many commands that access the target do not work when the target is running. Therefore, asynchronous command execution is most useful when combined with non-stop mode (see Section 5.5.2 [Non-Stop Mode], page 91). Then, it is possible to examine the state of one thread, while other threads are running.

When a given thread is running, MI commands that try to access the target in the context of that thread may not work, or may work only on some targets. In particular,

cli-command ↦
[*token*] *cli-command nl*, where *cli-command* is any existing GDB CLI command.

mi-command ↦
[*token*] "-" *operation* (" " *option*)* [" --"] (" " *parameter*)* *nl*

token ↦ "any sequence of digits"

option ↦ "-" *parameter* [" " *parameter*]

parameter ↦
non-blank-sequence | *c-string*

operation ↦
any of the operations described in this chapter

non-blank-sequence ↦
anything, provided it doesn't contain special characters such as "-", *nl,* """ *and of course* " "

c-string ↦
""" *seven-bit-iso-c-string-content* """

nl ↦ CR | CR-LF

Notes:

- The CLI commands are still handled by the MI interpreter; their output is described below.
- The *token*, when present, is passed back when the command finishes.
- Some MI commands accept optional arguments as part of the parameter list. Each option is identified by a leading '-' (dash) and may be followed by an optional argument parameter. Options occur first in the parameter list and can be delimited from normal parameters using '--' (this is useful when some parameters begin with a dash).

Pragmatics:

- We want easy access to the existing CLI syntax (for debugging).
- We want it to be easy to spot a MI operation.

27.4.2 GDB/MI Output Syntax

The output from GDB/MI consists of zero or more out-of-band records followed, optionally, by a single result record. This result record is for the most recent command. The sequence of output records is terminated by '(gdb)'.

If an input command was prefixed with a *token* then the corresponding output for that command will also be prefixed by that same *token*.

output ↦ (*out-of-band-record*)* [*result-record*] "(gdb)" *nl*

result-record ↦
[*token*] "^" *result-class* ("," *result*)* *nl*

out-of-band-record ↦
async-record | *stream-record*

- *status-async-output* contains on-going status information about the progress of a slow operation. It can be discarded. All status output is prefixed by '`+`'.
- *exec-async-output* contains asynchronous state change on the target (stopped, started, disappeared). All async output is prefixed by '`*`'.
- *notify-async-output* contains supplementary information that the client should handle (e.g., a new breakpoint information). All notify output is prefixed by '`=`'.
- *console-stream-output* is output that should be displayed as is in the console. It is the textual response to a CLI command. All the console output is prefixed by '`~`'.
- *target-stream-output* is the output produced by the target program. All the target output is prefixed by '`@`'.
- *log-stream-output* is output text coming from GDB's internals, for instance messages that should be displayed as part of an error log. All the log output is prefixed by '`&`'.
- New GDB/MI commands should only output *lists* containing *values*.

See Section 27.7.2 [GDB/MI Stream Records], page 545, for more details about the various output records.

27.5 GDB/MI Compatibility with CLI

For the developers convenience CLI commands can be entered directly, but there may be some unexpected behaviour. For example, commands that query the user will behave as if the user replied yes, breakpoint command lists are not executed and some CLI commands, such as `if`, `when` and `define`, prompt for further input with '`>`', which is not valid MI output.

This feature may be removed at some stage in the future and it is recommended that front ends use the `-interpreter-exec` command (see [-interpreter-exec], page 637).

27.6 GDB/MI Development and Front Ends

The application which takes the MI output and presents the state of the program being debugged to the user is called a *front end*.

Since GDB/MI is used by a variety of front ends to GDB, changes to the MI interface may break existing usage. This section describes how the protocol changes and how to request previous version of the protocol when it does.

Some changes in MI need not break a carefully designed front end, and for these the MI version will remain unchanged. The following is a list of changes that may occur within one level, so front ends should parse MI output in a way that can handle them:

- New MI commands may be added.
- New fields may be added to the output of any MI command.
- The range of values for fields with specified values, e.g., `in_scope` (see [-var-update], page 595) may be extended.

If the changes are likely to break front ends, the MI version level will be increased by one. The new versions of the MI protocol are not compatible with the old versions. Old versions of MI remain available, allowing front ends to keep using them until they are modified to use the latest MI version.

Since `--interpreter=mi` always points to the latest MI version, it is recommended that front ends request a specific version of MI when launching GDB (e.g. `--interpreter=mi2`) to make sure they get an interpreter with the MI version they expect.

The following table gives a summary of the released versions of the MI interface: the version number, the version of GDB in which it first appeared and the breaking changes compared to the previous version.

MI version	GDB version	Breaking changes
1	5.1	None
2	6.0	• The `-environment-pwd`, `-environment-directory` and `-environment-path` commands now returns values using the MI output syntax, rather than CLI output syntax. • `-var-list-children`'s `children` result field is now a list, rather than a tuple. • `-var-update`'s `changelist` result field is now a list, rather than a tuple.
3	9.1	• The output of information about multi-location breakpoints has changed in the responses to the `-break-insert` and `-break-info` commands, as well as in the `=breakpoint-created` and `=breakpoint-modified` events. The multiple locations are now placed in a `locations` field, whose value is a list.

If your front end cannot yet migrate to a more recent version of the MI protocol, you can nevertheless selectively enable specific features available in those recent MI versions, using the following commands:

`-fix-multi-location-breakpoint-output`
: Use the output for multi-location breakpoints which was introduced by MI 3, even when using MI versions 2 or 1. This command has no effect when using MI version 3 or later.

The best way to avoid unexpected changes in MI that might break your front end is to make your project known to GDB developers and follow development on `gdb@sourceware.org` and `gdb-patches@sourceware.org`.

27.7 GDB/MI Output Records

27.7.1 GDB/MI Result Records

In addition to a number of out-of-band notifications, the response to a GDB/MI command includes one of the following result indications:

`"^done" [ "," ` *`results`* ` ]`
: The synchronous operation was successful, *`results`* are the return values.

`"^running"`
: This result record is equivalent to '`^done`'. Historically, it was output instead of '`^done`' if the command has resumed the target. This behaviour is maintained for backward compatibility, but all frontends should treat '`^done`' and '`^running`' identically and rely on the '`*running`' output record to determine which threads are resumed.

`"^connected"`
: GDB has connected to a remote target.

`"^error" "," "msg=" c-string [ "," "code=" c-string ]`
: The operation failed. The `msg=`*`c-string`* variable contains the corresponding error message.

  If present, the `code=`*`c-string`* variable provides an error code on which consumers can rely on to detect the corresponding error condition. At present, only one error code is defined:

  '`"undefined-command"`'
  : Indicates that the command causing the error does not exist.

`"^exit"`
: GDB has terminated.

27.7.2 GDB/MI Stream Records

GDB internally maintains a number of output streams: the console, the target, and the log. The output intended for each of these streams is funneled through the GDB/MI interface using *stream records*.

Each stream record begins with a unique *prefix character* which identifies its stream (see Section 27.4.2 [GDB/MI Output Syntax], page 541). In addition to the prefix, each stream record contains a *`string-output`*. This is either raw text (with an implicit new line) or a quoted C string (which does not contain an implicit newline).

`"~" string-output`
: The console output stream contains text that should be displayed in the CLI console window. It contains the textual responses to CLI commands.

`"@" string-output`
: The target output stream contains any textual output from the running target. This is only present when GDB's event loop is truly asynchronous, which is currently only the case for remote targets.

`"&" string-output`
: The log stream contains debugging messages being produced by GDB's internals.

27.7.3 GDB/MI Async Records

Async records are used to notify the GDB/MI client of additional changes that have occurred. Those changes can either be a consequence of GDB/MI commands (e.g., a breakpoint modified) or a result of target activity (e.g., target stopped).

The following is the list of possible async records:

`*running,thread-id="thread"`
: The target is now running. The *thread* field can be the global thread ID of the thread that is now running, and it can be '`all`' if all threads are running. The

`vfork` The inferior has vforked. This is reported in when `catch vfork` (see Section 5.1.3 [Set Catchpoints], page 66) has been used.

`syscall-entry`
The inferior entered a system call. This is reported when `catch syscall` (see Section 5.1.3 [Set Catchpoints], page 66) has been used.

`syscall-return`
The inferior returned from a system call. This is reported when `catch syscall` (see Section 5.1.3 [Set Catchpoints], page 66) has been used.

`exec` The inferior called `exec`. This is reported when `catch exec` (see Section 5.1.3 [Set Catchpoints], page 66) has been used.

The *id* field identifies the global thread ID of the thread that directly caused the stop – for example by hitting a breakpoint. Depending on whether all-stop mode is in effect (see Section 5.5.1 [All-Stop Mode], page 90), GDB may either stop all threads, or only the thread that directly triggered the stop. If all threads are stopped, the *stopped* field will have the value of `"all"`. Otherwise, the value of the *stopped* field will be a list of thread identifiers. Presently, this list will always include a single thread, but frontend should be prepared to see several threads in the list. The *core* field reports the processor core on which the stop event has happened. This field may be absent if such information is not available.

`=thread-group-added,id="`*id*`"`
`=thread-group-removed,id="`*id*`"`
A thread group was either added or removed. The *id* field contains the GDB identifier of the thread group. When a thread group is added, it generally might not be associated with a running process. When a thread group is removed, its id becomes invalid and cannot be used in any way.

`=thread-group-started,id="`*id*`",pid="`*pid*`"`
A thread group became associated with a running program, either because the program was just started or the thread group was attached to a program. The *id* field contains the GDB identifier of the thread group. The *pid* field contains process identifier, specific to the operating system.

`=thread-group-exited,id="`*id*`"[,exit-code="`*code*`"]`
A thread group is no longer associated with a running program, either because the program has exited, or because it was detached from. The *id* field contains the GDB identifier of the thread group. The *code* field is the exit code of the inferior; it exists only when the inferior exited with some code.

`=thread-created,id="`*id*`",group-id="`*gid*`"`
`=thread-exited,id="`*id*`",group-id="`*gid*`"`
A thread either was created, or has exited. The *id* field contains the global GDB identifier of the thread. The *gid* field identifies the thread group this thread belongs to.

`=tsv-modified,name=`*name*`,initial=`*initial*`[,current=`*current*`]`
: Reports that the trace state variable *name* is modified with the initial value *initial*. The current value *current* of trace state variable is optional and is reported if the current value of trace state variable is known.

`=breakpoint-created,bkpt={...}`
`=breakpoint-modified,bkpt={...}`
`=breakpoint-deleted,id=`*number*
: Reports that a breakpoint was created, modified, or deleted, respectively. Only user-visible breakpoints are reported to the MI user.

  The *bkpt* argument is of the same form as returned by the various breakpoint commands; See Section 27.10 [GDB/MI Breakpoint Commands], page 554. The *number* is the ordinal number of the breakpoint.

  Note that if a breakpoint is emitted in the result record of a command, then it will not also be emitted in an async record.

`=record-started,thread-group="`*id*`",method="`*method*`"[,format="`*format*`"]`
`=record-stopped,thread-group="`*id*`"`
: Execution log recording was either started or stopped on an inferior. The *id* is the GDB identifier of the thread group corresponding to the affected inferior.

  The *method* field indicates the method used to record execution. If the method in use supports multiple recording formats, *format* will be present and contain the currently used format. See Chapter 7 [Process Record and Replay], page 99, for existing method and format values.

`=cmd-param-changed,param=`*param*`,value=`*value*
: Reports that a parameter of the command `set` *param* is changed to *value*. In the multi-word `set` command, the *param* is the whole parameter list to `set` command. For example, In command `set check type on`, *param* is `check type` and *value* is `on`.

`=memory-changed,thread-group=`*id*`,addr=`*addr*`,len=`*len*`[,type="code"]`
: Reports that bytes from *addr* to *data* + *len* were written in an inferior. The *id* is the identifier of the thread group corresponding to the affected inferior. The optional `type="code"` part is reported if the memory written to holds executable code.

27.7.4 GDB/MI Breakpoint Information

When GDB reports information about a breakpoint, a tracepoint, a watchpoint, or a catchpoint, it uses a tuple with the following fields:

`number`
: The breakpoint number.

`type`
: The type of the breakpoint. For ordinary breakpoints this will be '`breakpoint`', but many values are possible.

`catch-type`
: If the type of the breakpoint is '`catchpoint`', then this indicates the exact type of catchpoint.

`disp` This is the breakpoint disposition—either '`del`', meaning that the breakpoint will be deleted at the next stop, or '`keep`', meaning that the breakpoint will not be deleted.

`enabled` This indicates whether the breakpoint is enabled, in which case the value is '`y`', or disabled, in which case the value is '`n`'. Note that this is not the same as the field `enable`.

`addr` The address of the breakpoint. This may be a hexidecimal number, giving the address; or the string '`<PENDING>`', for a pending breakpoint; or the string '`<MULTIPLE>`', for a breakpoint with multiple locations. This field will not be present if no address can be determined. For example, a watchpoint does not have an address.

`addr_flags`
Optional field containing any flags related to the address. These flags are architecture-dependent; see Section 21.4 [Architectures], page 338, for their meaning for a particular CPU.

`func` If known, the function in which the breakpoint appears. If not known, this field is not present.

`filename` The name of the source file which contains this function, if known. If not known, this field is not present.

`fullname` The full file name of the source file which contains this function, if known. If not known, this field is not present.

`line` The line number at which this breakpoint appears, if known. If not known, this field is not present.

`at` If the source file is not known, this field may be provided. If provided, this holds the address of the breakpoint, possibly followed by a symbol name.

`pending` If this breakpoint is pending, this field is present and holds the text used to set the breakpoint, as entered by the user.

`evaluated-by`
Where this breakpoint's condition is evaluated, either '`host`' or '`target`'.

`thread` If this is a thread-specific breakpoint, then this identifies the thread in which the breakpoint can trigger.

`task` If this breakpoint is restricted to a particular Ada task, then this field will hold the task identifier.

`cond` If the breakpoint is conditional, this is the condition expression.

`ignore` The ignore count of the breakpoint.

`enable` The enable count of the breakpoint.

`traceframe-usage`
FIXME.

`static-tracepoint-marker-string-id`
For a static tracepoint, the name of the static tracepoint marker.

For example, here is what the output of `-break-insert` (see Section 27.10 [GDB/MI Breakpoint Commands], page 554) might be:

```
-> -break-insert main
<- ^done,bkpt={number="1",type="breakpoint",disp="keep",
    enabled="y",addr="0x08048564",func="main",file="myprog.c",
    fullname="/home/nickrob/myprog.c",line="68",thread-groups=["i1"],
    times="0"}
<- (gdb)
```

27.7.5 GDB/MI Frame Information

Response from many MI commands includes an information about stack frame. This information is a tuple that may have the following fields:

`level` The level of the stack frame. The innermost frame has the level of zero. This field is always present.

`func` The name of the function corresponding to the frame. This field may be absent if GDB is unable to determine the function name.

`addr` The code address for the frame. This field is always present.

`addr_flags`
Optional field containing any flags related to the address. These flags are architecture-dependent; see Section 21.4 [Architectures], page 338, for their meaning for a particular CPU.

`file` The name of the source files that correspond to the frame's code address. This field may be absent.

`line` The source line corresponding to the frames' code address. This field may be absent.

`from` The name of the binary file (either executable or shared library) the corresponds to the frame's code address. This field may be absent.

27.7.6 GDB/MI Thread Information

Whenever GDB has to report an information about a thread, it uses a tuple with the following fields. The fields are always present unless stated otherwise.

`id` The global numeric id assigned to the thread by GDB.

`target-id`
The target-specific string identifying the thread.

`details` Additional information about the thread provided by the target. It is supposed to be human-readable and not interpreted by the frontend. This field is optional.

`name` The name of the thread. If the user specified a name using the `thread name` command, then this name is given. Otherwise, if GDB can extract the thread name from the target, then that name is given. If GDB cannot find the thread name, then this field is omitted.

`state` The execution state of the thread, either '`stopped`' or '`running`', depending on whether the thread is presently running.

`frame` The stack frame currently executing in the thread. This field is only present if the thread is stopped. Its format is documented in Section 27.7.5 [GDB/MI Frame Information], page 552.

`core` The value of this field is an integer number of the processor core the thread was last seen on. This field is optional.

27.7.7 GDB/MI Ada Exception Information

Whenever a `*stopped` record is emitted because the program stopped after hitting an exception catchpoint (see Section 5.1.3 [Set Catchpoints], page 66), GDB provides the name of the exception that was raised via the `exception-name` field. Also, for exceptions that were raised with an exception message, GDB provides that message via the `exception-message` field.

27.8 Simple Examples of GDB/MI Interaction

This subsection presents several simple examples of interaction using the GDB/MI interface. In these examples, '`->`' means that the following line is passed to GDB/MI as input, while '`<-`' means the output received from GDB/MI.

Note the line breaks shown in the examples are here only for readability, they don't appear in the real output.

Setting a Breakpoint

Setting a breakpoint generates synchronous output which contains detailed information of the breakpoint.

```
-> -break-insert main
<- ^done,bkpt={number="1",type="breakpoint",disp="keep",
    enabled="y",addr="0x08048564",func="main",file="myprog.c",
    fullname="/home/nickrob/myprog.c",line="68",thread-groups=["i1"],
    times="0"}
<- (gdb)
```

Program Execution

Program execution generates asynchronous records and MI gives the reason that execution stopped.

```
-> -exec-run
<- ^running
<- (gdb)
<- *stopped,reason="breakpoint-hit",disp="keep",bkptno="1",thread-id="0",
   frame={addr="0x08048564",func="main",
   args=[{name="argc",value="1"},{name="argv",value="0xbfc4d4d4"}],
   file="myprog.c",fullname="/home/nickrob/myprog.c",line="68",
   arch="i386:x86_64"}
<- (gdb)
-> -exec-continue
<- ^running
<- (gdb)
<- *stopped,reason="exited-normally"
<- (gdb)
```

Quitting GDB

Quitting GDB just prints the result class '`^exit`'.

```
-> (gdb)
<- -gdb-exit
<- ^exit
```

Please note that '`^exit`' is printed immediately, but it might take some time for GDB to actually exit. During that time, GDB performs necessary cleanups, including killing programs being debugged or disconnecting from debug hardware, so the frontend should wait till GDB exits and should only forcibly kill GDB if it fails to exit in reasonable time.

A Bad Command

Here's what happens if you pass a non-existent command:

```
-> -rubbish
<- ^error,msg="Undefined MI command: rubbish"
<- (gdb)
```

27.9 GDB/MI Command Description Format

The remaining sections describe blocks of commands. Each block of commands is laid out in a fashion similar to this section.

Motivation

The motivation for this collection of commands.

Introduction

A brief introduction to this collection of commands as a whole.

Commands

For each command in the block, the following is described:

Synopsis

`-command` *args...*

Result

GDB Command

The corresponding GDB CLI command(s), if any.

Example

Example(s) formatted for readability. Some of the described commands have not been implemented yet and these are labeled N.A. (not available).

27.10 GDB/MI Breakpoint Commands

This section documents GDB/MI commands for manipulating breakpoints.

The `-break-after` Command

```
enabled="y",addr="0x000100d0",func="main",file="hello.c",
fullname="/home/foo/hello.c",line="5",thread-groups=["i1"],
times="0"}
(gdb)
-break-commands 1 "print v" "continue"
^done
(gdb)
```

The `-break-condition` Command

Synopsis

```
-break-condition [ --force ] number [ expr ]
```

Breakpoint *number* will stop the program only if the condition in *expr* is true. The condition becomes part of the '`-break-list`' output (see the description of the '`-break-list`' command below). If the '`--force`' flag is passed, the condition is forcibly defined even when it is invalid for all locations of breakpoint *number*. If the *expr* argument is omitted, breakpoint *number* becomes unconditional.

GDB Command

The corresponding GDB command is '`condition`'.

Example

```
(gdb)
-break-condition 1 1
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",fullname="/home/foo/hello.c",
line="5",cond="1",thread-groups=["i1"],times="0",ignore="3"}]}
(gdb)
```

The `-break-delete` Command

Synopsis

```
-break-delete ( breakpoint )+
```

Delete the breakpoint(s) whose number(s) are specified in the argument list. This is obviously reflected in the breakpoint list.

GDB Command

The corresponding GDB command is '`delete`'.

Example

```
(gdb)
```

```
-break-delete 1
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="0",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[]}
(gdb)
```

The `-break-disable` Command

Synopsis

```
-break-disable ( breakpoint )+
```

Disable the named *breakpoint*(s). The field '`enabled`' in the break list is now set to '`n`' for the named *breakpoint*(s).

GDB Command

The corresponding GDB command is '`disable`'.

Example

```
(gdb)
-break-disable 2
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="2",type="breakpoint",disp="keep",enabled="n",
addr="0x000100d0",func="main",file="hello.c",fullname="/home/foo/hello.c",
line="5",thread-groups=["i1"],times="0"}]}
(gdb)
```

The `-break-enable` Command

Synopsis

```
-break-enable ( breakpoint )+
```

Enable (previously disabled) *breakpoint*(s).

GDB Command

The corresponding GDB command is '`enable`'.

Example

```
(gdb)
```

```
-break-enable 2
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="2",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",fullname="/home/foo/hello.c",
line="5",thread-groups=["i1"],times="0"}]}
(gdb)
```

The `-break-info` Command

Synopsis

```
-break-info breakpoint
```

Get information about a single breakpoint.

The result is a table of breakpoints. See Section 27.7.4 [GDB/MI Breakpoint Information], page 549, for details on the format of each breakpoint in the table.

GDB Command

The corresponding GDB command is '`info break` *`breakpoint`*'.

Example

N.A.

The `-break-insert` Command

Synopsis

```
-break-insert [ -t ] [ -h ] [ -f ] [ -d ] [ -a ] [ --qualified ]
   [ -c condition ] [ --force-condition ] [ -i ignore-count ]
```

‘`--label` *label*’
: The name of a label.

‘`--line` *lineoffset*’
: An absolute or relative line offset from the start of the location.

address location
: An address location, **address*. See Section 9.2.3 [Address Locations], page 122.

The possible optional parameters of this command are:

‘`-t`’ Insert a temporary breakpoint.

‘`-h`’ Insert a hardware breakpoint.

‘`-f`’ If *location* cannot be parsed (for example if it refers to unknown files or functions), create a pending breakpoint. Without this flag, GDB will report an error, and won't create a breakpoint, if *location* cannot be parsed.

‘`-d`’ Create a disabled breakpoint.

‘`-a`’ Create a tracepoint. See Chapter 13 [Tracepoints], page 193. When this parameter is used together with ‘`-h`’, a fast tracepoint is created.

‘`-c` *condition*’
: Make the breakpoint conditional on *condition*.

‘`--force-condition`’
: Forcibly define the breakpoint even if the condition is invalid at all of the breakpoint locations.

‘`-i` *ignore-count*’
: Initialize the *ignore-count*.

‘`-p` *thread-id*’
: Restrict the breakpoint to the thread with the specified global *thread-id*.

‘`--qualified`’
: This option makes GDB interpret a function name specified as a complete fully-qualified name.

Result

See Section 27.7.4 [GDB/MI Breakpoint Information], page 549, for details on the format of the resulting breakpoint.

Note: this format is open to change.

GDB Command

The corresponding GDB commands are ‘`break`’, ‘`tbreak`’, ‘`hbreak`’, and ‘`thbreak`’.

Example

```
(gdb)
-break-insert main
^done,bkpt={number="1",addr="0x0001072c",file="recursive2.c",
fullname="/home/foo/recursive2.c,line="4",thread-groups=["i1"],
```

Result

See Section 27.7.4 [GDB/MI Breakpoint Information], page 549, for details on the format of the resulting breakpoint.

GDB Command

The corresponding GDB command is '`dprintf`'.

Example

```
(gdb)
4-dprintf-insert foo "At foo entry\n"
4^done,bkpt={number="1",type="dprintf",disp="keep",enabled="y",
addr="0x000000000040061b",func="foo",file="mi-dprintf.c",
fullname="mi-dprintf.c",line="25",thread-groups=["i1"],
times="0",script={"printf \"At foo entry\\n\"","continue"},
original-location="foo"}
(gdb)
5-dprintf-insert 26 "arg=%d, g=%d\n" arg g
5^done,bkpt={number="2",type="dprintf",disp="keep",enabled="y",
addr="0x000000000040062a",func="foo",file="mi-dprintf.c",
fullname="mi-dprintf.c",line="26",thread-groups=["i1"],
times="0",script={"printf \"arg=%d, g=%d\\n\", arg, g","continue"},
original-location="mi-dprintf.c:26"}
(gdb)
```

The `-break-list` Command

Synopsis

```
-break-list
```

Displays the list of inserted breakpoints, showing the following fields:

'`Number`' number of the breakpoint

'`Type`' type of the breakpoint: '`breakpoint`' or '`watchpoint`'

'`Disposition`'
should the breakpoint be deleted or disabled when it is hit: '`keep`' or '`nokeep`'

'`Enabled`' is the breakpoint enabled or no: '`y`' or '`n`'

'`Address`' memory location at which the breakpoint is set

'`What`' logical location of the breakpoint, expressed by function name, file name, line number

'`Thread-groups`'
list of thread groups to which this breakpoint applies

'`Times`' number of times the breakpoint has been hit

If there are no breakpoints or watchpoints, the `BreakpointTable` `body` field is an empty list.

GDB Command

The corresponding GDB command is '`info break`'.

Example

```
(gdb)
-break-list
^done,BreakpointTable={nr_rows="2",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",thread-groups=["i1"],
times="0"},
bkpt={number="2",type="breakpoint",disp="keep",enabled="y",
addr="0x00010114",func="foo",file="hello.c",fullname="/home/foo/hello.c",
line="13",thread-groups=["i1"],times="0"}]}
(gdb)
```

Here's an example of the result when there are no breakpoints:

```
(gdb)
-break-list
^done,BreakpointTable={nr_rows="0",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[]}
(gdb)
```

The `-break-passcount` Command

Synopsis

```
-break-passcount tracepoint-number passcount
```

Set the passcount for tracepoint *tracepoint-number* to *passcount*. If the breakpoint referred to by *tracepoint-number* is not a tracepoint, error is emitted. This corresponds to CLI command '`passcount`'.

The `-break-watch` Command

Synopsis

```
-break-watch [ -a | -r ]
```

Create a watchpoint. With the '`-a`' option it will create an *access* watchpoint, i.e., a watchpoint that triggers either on a read from or on a write to the memory location. With the '`-r`' option, the watchpoint created is a *read* watchpoint, i.e., it will trigger only when the memory location is accessed for reading. Without either of the options, the watchpoint created is a regular watchpoint, i.e., it will trigger when the memory location is accessed for writing. See Section 5.1.2 [Setting Watchpoints], page 63.

Note that '`-break-list`' will report a single list of watchpoints and breakpoints inserted.

GDB Command

The corresponding GDB commands are '`watch`', '`awatch`', and '`rwatch`'.

Example

Setting a watchpoint on a variable in the `main` function:

```
(gdb)
-break-watch x
^done,wpt={number="2",exp="x"}
(gdb)
-exec-continue
^running
(gdb)
*stopped,reason="watchpoint-trigger",wpt={number="2",exp="x"},
value={old="-268439212",new="55"},
frame={func="main",args=[],file="recursive2.c",
fullname="/home/foo/bar/recursive2.c",line="5",arch="i386:x86_64"}
(gdb)
```

Setting a watchpoint on a variable local to a function. GDB will stop the program execution twice: first for the variable changing value, then for the watchpoint going out of scope.

```
(gdb)
-break-watch C
^done,wpt={number="5",exp="C"}
(gdb)
-exec-continue
^running
(gdb)
*stopped,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",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="13",
arch="i386:x86_64"}
(gdb)
-exec-continue
^running
(gdb)
*stopped,reason="watchpoint-scope",wpnum="5",
frame={func="callee3",args=[{name="strarg",
value="0x11940 \"A string argument.\""}],
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="18",
arch="i386:x86_64"}
(gdb)
```

Listing breakpoints and watchpoints, at different points in the program execution. Note that once the watchpoint goes out of scope, it is deleted.

```
(gdb)
-break-watch C
^done,wpt={number="2",exp="C"}
(gdb)
-break-list
^done,BreakpointTable={nr_rows="2",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
```

```
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/devo/gdb/testsuite/gdb.mi/basics.c"line="8",thread-groups=["i1"],
times="1"},
bkpt={number="2",type="watchpoint",disp="keep",
enabled="y",addr="",what="C",thread-groups=["i1"],times="0"}]}
(gdb)
-exec-continue
^running
(gdb)
*stopped,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",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="13",
arch="i386:x86_64"}
(gdb)
-break-list
^done,BreakpointTable={nr_rows="2",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/devo/gdb/testsuite/gdb.mi/basics.c",line="8",thread-groups=["i1"],
times="1"},
bkpt={number="2",type="watchpoint",disp="keep",
enabled="y",addr="",what="C",thread-groups=["i1"],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",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="18",
arch="i386:x86_64"}
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/devo/gdb/testsuite/gdb.mi/basics.c",line="8",
thread-groups=["i1"],times="1"}]}
(gdb)
```

27.11 GDB/MI Catchpoint Commands

This section documents GDB/MI commands for manipulating catchpoints.

27.11.1 Shared Library GDB/MI Catchpoints

The `-catch-load` Command

Synopsis

```
-catch-load [ -t ] [ -d ] regexp
```

Add a catchpoint for library load events. If the '`-t`' option is used, the catchpoint is a temporary one (see Section 5.1.1 [Setting Breakpoints], page 56). If the '`-d`' option is used, the catchpoint is created in a disabled state. The '`regexp`' argument is a regular expression used to match the name of the loaded library.

GDB Command

The corresponding GDB command is '`catch load`'.

Example

```
-catch-load -t foo.so
^done,bkpt={number="1",type="catchpoint",disp="del",enabled="y",
what="load of library matching foo.so",catch-type="load",times="0"}
(gdb)
```

The `-catch-unload` Command

Synopsis

```
-catch-unload [ -t ] [ -d ] regexp
```

Add a catchpoint for library unload events. If the '`-t`' option is used, the catchpoint is a temporary one (see Section 5.1.1 [Setting Breakpoints], page 56). If the '`-d`' option is used, the catchpoint is created in a disabled state. The '`regexp`' argument is a regular expression used to match the name of the unloaded library.

GDB Command

The corresponding GDB command is '`catch unload`'.

Example

```
-catch-unload -d bar.so
^done,bkpt={number="2",type="catchpoint",disp="keep",enabled="n",
what="load of library matching bar.so",catch-type="unload",times="0"}
(gdb)
```

27.11.2 Ada Exception GDB/MI Catchpoints

The following GDB/MI commands can be used to create catchpoints that stop the execution when Ada exceptions are being raised.

The `-catch-assert` Command

Synopsis

```
-catch-assert [ -c condition] [ -d ] [ -t ]
```

Add a catchpoint for failed Ada assertions.

The possible optional parameters for this command are:

'`-c condition`'
: Make the catchpoint conditional on *condition*.

'`-d`'
: Create a disabled catchpoint.

'`-t`'
: Create a temporary catchpoint.

GDB Command

The corresponding GDB command is '`catch assert`'.

Example

```
-catch-assert
^done,bkptno="5",bkpt={number="5",type="breakpoint",disp="keep",
enabled="y",addr="0x0000000000404888",what="failed Ada assertions",
thread-groups=["i1"],times="0",
original-location="__gnat_debug_raise_assert_failure"}
(gdb)
```

The `-catch-exception` Command

Synopsis

```
-catch-exception [ -c condition] [ -d ] [ -e exception-name ]
   [ -t ] [ -u ]
```

Add a catchpoint stopping when Ada exceptions are raised. By default, the command stops the program when any Ada exception gets raised. But it is also possible, by using some of the optional parameters described below, to create more selective catchpoints.

The possible optional parameters for this command are:

'`-c condition`'
: Make the catchpoint conditional on *condition*.

'`-d`'
: Create a disabled catchpoint.

'`-e exception-name`'
: Only stop when *exception-name* is raised. This option cannot be used combined with '`-u`'.

'`-t`'
: Create a temporary catchpoint.

'`-u`'
: Stop only when an unhandled exception gets raised. This option cannot be used combined with '`-e`'.

GDB Command

The corresponding GDB commands are '`catch exception`' and '`catch exception unhandled`'.

Example

```
-catch-exception -e Program_Error
^done,bkptno="4",bkpt={number="4",type="breakpoint",disp="keep",
enabled="y",addr="0x0000000000404874",
what="‘Program_Error’ Ada exception", thread-groups=["i1"],
times="0",original-location="__gnat_debug_raise_exception"}
(gdb)
```

The `-catch-handlers` Command

Synopsis

```
-catch-handlers [ -c condition] [ -d ] [ -e exception-name ]
   [ -t ]
```

Add a catchpoint stopping when Ada exceptions are handled. By default, the command stops the program when any Ada exception gets handled. But it is also possible, by using some of the optional parameters described below, to create more selective catchpoints.

The possible optional parameters for this command are:

‘`-c` *condition*’
: Make the catchpoint conditional on *condition*.

‘`-d`’
: Create a disabled catchpoint.

‘`-e` *exception-name*’
: Only stop when *exception-name* is handled.

‘`-t`’
: Create a temporary catchpoint.

GDB Command

The corresponding GDB command is ‘`catch handlers`’.

Example

```
-catch-handlers -e Constraint_Error
^done,bkptno="4",bkpt={number="4",type="breakpoint",disp="keep",
enabled="y",addr="0x0000000000402f68",
what="‘Constraint_Error’ Ada exception handlers",thread-groups=["i1"],
times="0",original-location="__gnat_begin_handler"}
(gdb)
```

27.11.3 C++ Exception GDB/MI Catchpoints

The following GDB/MI commands can be used to create catchpoints that stop the execution when C++ exceptions are being throw, rethrown, or caught.

The `-catch-throw` Command

Synopsis

```
-catch-throw [ -t ] [ -r regexp]
```

Stop when the debuggee throws a C++ exception. If *regexp* is given, then only exceptions whose type matches the regular expression will be caught.

If ‘`-t`’ is given, then the catchpoint is enabled only for one stop, the catchpoint is automatically deleted after stopping once for the event.

GDB Command

The corresponding GDB commands are '`catch throw`' and '`tcatch throw`' (see Section 5.1.3 [Set Catchpoints], page 66).

Example

```
-catch-throw -r exception_type
^done,bkpt={number="1",type="catchpoint",disp="keep",enabled="y",
  what="exception throw",catch-type="throw",
  thread-groups=["i1"],
  regexp="exception_type",times="0"}
(gdb)
-exec-run
^running
(gdb)
~"\n"
~"Catchpoint 1 (exception thrown), 0x00007ffff7ae00ed
  in __cxa_throw () from /lib64/libstdc++.so.6\n"
*stopped,bkptno="1",reason="breakpoint-hit",disp="keep",
  frame={addr="0x00007ffff7ae00ed",func="__cxa_throw",
  args=[],from="/lib64/libstdc++.so.6",arch="i386:x86-64"},
  thread-id="1",stopped-threads="all",core="6"
(gdb)
```

The `-catch-rethrow` Command

Synopsis

```
-catch-rethrow [ -t ] [ -r regexp]
```

Stop when a C++ exception is re-thrown. If *regexp* is given, then only exceptions whose type matches the regular expression will be caught.

If '`-t`' is given, then the catchpoint is enabled only for one stop, the catchpoint is automatically deleted after the first event is caught.

GDB Command

The corresponding GDB commands are '`catch rethrow`' and '`tcatch rethrow`' (see Section 5.1.3 [Set Catchpoints], page 66).

Example

```
-catch-rethrow -r exception_type
^done,bkpt={number="1",type="catchpoint",disp="keep",enabled="y",
  what="exception rethrow",catch-type="rethrow",
  thread-groups=["i1"],
  regexp="exception_type",times="0"}
(gdb)
-exec-run
^running
(gdb)
~"\n"
~"Catchpoint 1 (exception rethrown), 0x00007ffff7ae00ed
  in __cxa_rethrow () from /lib64/libstdc++.so.6\n"
*stopped,bkptno="1",reason="breakpoint-hit",disp="keep",
  frame={addr="0x00007ffff7ae00ed",func="__cxa_rethrow",
  args=[],from="/lib64/libstdc++.so.6",arch="i386:x86-64"},
```

The -environment-cd Command

Synopsis

```
-environment-cd pathdir
```

Set GDB's working directory.

GDB Command

The corresponding GDB command is '`cd`'.

Example

```
(gdb)
-environment-cd /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done
(gdb)
```

The -environment-directory Command

Synopsis

```
-environment-directory [ -r ] [ pathdir ]+
```

Add directories *pathdir* to beginning of search path for source files. If the '`-r`' option is used, the search path is reset to the default search path. If directories *pathdir* are supplied in addition to the '`-r`' option, the search path is first reset and then addition occurs as normal. Multiple directories may be specified, separated by blanks. Specifying multiple directories in a single command results in the directories added to the beginning of the search path in the same order they were presented in the command. If blanks are needed as part of a directory name, double-quotes should be used around the name. In the command output, the path will show up separated by the system directory-separator character. The directory-separator character must not be used in any directory name. If no directories are specified, the current search path is displayed.

GDB Command

The corresponding GDB command is '`dir`'.

Example

```
(gdb)
-environment-directory /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done,source-path="/kwikemart/marge/ezannoni/flathead-dev/devo/gdb:$cdir:$cwd"
(gdb)
-environment-directory ""
^done,source-path="/kwikemart/marge/ezannoni/flathead-dev/devo/gdb:$cdir:$cwd"
(gdb)
-environment-directory -r /home/jjohnstn/src/gdb /usr/src
^done,source-path="/home/jjohnstn/src/gdb:/usr/src:$cdir:$cwd"
(gdb)
-environment-directory -r
^done,source-path="$cdir:$cwd"
(gdb)
```

The -environment-path Command

Synopsis

```
-environment-path [ -r ] [ pathdir ]+
```

Add directories *pathdir* to beginning of search path for object files. If the '`-r`' option is used, the search path is reset to the original search path that existed at gdb start-up. If directories *pathdir* are supplied in addition to the '`-r`' option, the search path is first reset and then addition occurs as normal. Multiple directories may be specified, separated by blanks. Specifying multiple directories in a single command results in the directories added to the beginning of the search path in the same order they were presented in the command. If blanks are needed as part of a directory name, double-quotes should be used around the name. In the command output, the path will show up separated by the system directory-separator character. The directory-separator character must not be used in any directory name. If no directories are specified, the current path is displayed.

GDB Command

The corresponding GDB command is '`path`'.

Example

```
(gdb)
-environment-path
^done,path="/usr/bin"
(gdb)
-environment-path /kwikemart/marge/ezannoni/flathead-dev/ppc-eabi/gdb /bin
^done,path="/kwikemart/marge/ezannoni/flathead-dev/ppc-eabi/gdb:/bin:/usr/bin"
(gdb)
-environment-path -r /usr/local/bin
^done,path="/usr/local/bin:/usr/bin"
(gdb)
```

The `-environment-pwd` Command

Synopsis

```
-environment-pwd
```

Show the current working directory.

GDB Command

The corresponding GDB command is '`pwd`'.

Example

```
(gdb)
-environment-pwd
^done,cwd="/kwikemart/marge/ezannoni/flathead-dev/devo/gdb"
(gdb)
```

27.13 GDB/MI Thread Commands

The `-thread-info` Command

Synopsis

```
-thread-info [ thread-id ]
```

Reports information about either a specific thread, if the *thread-id* parameter is present, or about all threads. *thread-id* is the thread's global thread ID. When printing information about all threads, also reports the global ID of the current thread.

GDB Command

The '`info thread`' command prints the same information about all threads.

Result

The result contains the following attributes:

'`threads`' A list of threads. The format of the elements of the list is described in Section 27.7.6 [GDB/MI Thread Information], page 552.

'`current-thread-id`'
: The global id of the currently selected thread. This field is omitted if there is no selected thread (for example, when the selected inferior is not running, and therefore has no threads) or if a *thread-id* argument was passed to the command.

Example

```
-thread-info
^done,threads=[
{id="2",target-id="Thread 0xb7e14b90 (LWP 21257)",
   frame={level="0",addr="0xffffe410",func="__kernel_vsyscall",
          args=[]},state="running"},
{id="1",target-id="Thread 0xb7e156b0 (LWP 21254)",
   frame={level="0",addr="0x0804891f",func="foo",
          args=[{name="i",value="10"}],
          file="/tmp/a.c",fullname="/tmp/a.c",line="158",arch="i386:x86_64"},
          state="running"}],
current-thread-id="1"
(gdb)
```

The `-thread-list-ids` Command

Synopsis

```
-thread-list-ids
```

Produces a list of the currently known global GDB thread ids. At the end of the list it also prints the total number of such threads.

This command is retained for historical reasons, the `-thread-info` command should be used instead.

GDB Command

Part of '`info threads`' supplies the same information.

Example

```
(gdb)
-thread-list-ids
^done,thread-ids={thread-id="3",thread-id="2",thread-id="1"},
current-thread-id="1",number-of-threads="3"
(gdb)
```

The `-thread-select` Command

Synopsis

```
-thread-select thread-id
```

Make thread with global thread number *thread-id* the current thread. It prints the number of the new current thread, and the topmost frame for that thread.

This command is deprecated in favor of explicitly using the '`--thread`' option to each command.

GDB Command

The corresponding GDB command is '`thread`'.

Example

```
(gdb)
-exec-next
^running
(gdb)
*stopped,reason="end-stepping-range",thread-id="2",line="187",
file="../../../devo/gdb/testsuite/gdb.threads/linux-dp.c"
(gdb)
-thread-list-ids
^done,
thread-ids={thread-id="3",thread-id="2",thread-id="1"},
number-of-threads="3"
(gdb)
-thread-select 3
^done,new-thread-id="3",
frame={level="0",func="vprintf",
args=[{name="format",value="0x8048e9c \"%*s%c %d %c\\n\""},
{name="arg",value="0x2"}],file="vprintf.c",line="31",arch="i386:x86_64"}
(gdb)
```

27.14 GDB/MI Ada Tasking Commands

The `-ada-task-info` Command

Synopsis

```
-ada-task-info [ task-id ]
```

Reports information about either a specific Ada task, if the *task-id* parameter is present, or about all Ada tasks.

GDB Command

The '`info tasks`' command prints the same information about all Ada tasks (see Section 15.4.10.7 [Ada Tasks], page 243).

Result

The result is a table of Ada tasks. The following columns are defined for each Ada task:

'`current`' This field exists only for the current thread. It has the value '`*`'.

'`id`' The identifier that GDB uses to refer to the Ada task.

'`task-id`' The identifier that the target uses to refer to the Ada task.

'`thread-id`'
The global thread identifier of the thread corresponding to the Ada task.

This field should always exist, as Ada tasks are always implemented on top of a thread. But if GDB cannot find this corresponding thread for any reason, the field is omitted.

'`parent-id`'
This field exists only when the task was created by another task. In this case, it provides the ID of the parent task.

'`priority`'
The base priority of the task.

'`state`' The current state of the task. For a detailed description of the possible states, see Section 15.4.10.7 [Ada Tasks], page 243.

'`name`' The name of the task.

Example

```
-ada-task-info
^done,tasks={nr_rows="3",nr_cols="8",
hdr=[{width="1",alignment="-1",col_name="current",colhdr=""},
{width="3",alignment="1",col_name="id",colhdr="ID"},
{width="9",alignment="1",col_name="task-id",colhdr="TID"},
{width="4",alignment="1",col_name="thread-id",colhdr=""},
{width="4",alignment="1",col_name="parent-id",colhdr="P-ID"},
{width="3",alignment="1",col_name="priority",colhdr="Pri"},
{width="22",alignment="-1",col_name="state",colhdr="State"},
{width="1",alignment="2",col_name="name",colhdr="Name"}],
body=[{current="*",id="1",task-id="   644010",thread-id="1",priority="48",
state="Child Termination Wait",name="main_task"}]}
(gdb)
```

27.15 GDB/MI Program Execution

These are the asynchronous commands which generate the out-of-band record '`*stopped`'. Currently GDB only really executes asynchronously with remote targets and this interaction is mimicked in other cases.

The `-exec-continue` Command

Synopsis

```
-exec-continue [--reverse] [--all|--thread-group N]
```

Resumes the execution of the inferior program, which will continue to execute until it reaches a debugger stop event. If the '`--reverse`' option is specified, execution resumes in reverse until it reaches a stop event. Stop events may include

- breakpoints or watchpoints
- signals or exceptions

- the end of the process (or its beginning under '`--reverse`')
- the end or beginning of a replay log if one is being used.

In all-stop mode (see Section 5.5.1 [All-Stop Mode], page 90), may resume only one thread, or all threads, depending on the value of the '`scheduler-locking`' variable. If '`--all`' is specified, all threads (in all inferiors) will be resumed. The '`--all`' option is ignored in all-stop mode. If the '`--thread-group`' options is specified, then all threads in that thread group are resumed.

GDB Command

The corresponding GDB corresponding is '`continue`'.

Example

```
-exec-continue
^running
(gdb)
@Hello world
*stopped,reason="breakpoint-hit",disp="keep",bkptno="2",frame={
func="foo",args=[],file="hello.c",fullname="/home/foo/bar/hello.c",
line="13",arch="i386:x86_64"}
(gdb)
```

The `-exec-finish` Command

Synopsis

```
-exec-finish [--reverse]
```

Resumes the execution of the inferior program until the current function is exited. Displays the results returned by the function. If the '`--reverse`' option is specified, resumes the reverse execution of the inferior program until the point where current function was called.

GDB Command

The corresponding GDB command is '`finish`'.

Example

Function returning `void`.

```
-exec-finish
^running
(gdb)
@hello from foo
*stopped,reason="function-finished",frame={func="main",args=[],
file="hello.c",fullname="/home/foo/bar/hello.c",line="7",arch="i386:x86_64"}
(gdb)
```

Function returning other than `void`. The name of the internal GDB variable storing the result is printed, together with the value itself.

```
-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",fullname="/home/foo/bar/recursive2.c",line="14",
arch="i386:x86_64"},
gdb-result-var="$1",return-value="0"
(gdb)
```

The -exec-interrupt Command

Synopsis

```
-exec-interrupt [--all|--thread-group N]
```

Interrupts the background execution of the target. Note how the token associated with the stop message is the one for the execution command that has been interrupted. The token for the interrupt itself only appears in the '`^done`' output. If the user is trying to interrupt a non-running program, an error message will be printed.

Note that when asynchronous execution is enabled, this command is asynchronous just like other execution commands. That is, first the '`^done`' response will be printed, and the target stop will be reported after that using the '`*stopped`' notification.

In non-stop mode, only the context thread is interrupted by default. All threads (in all inferiors) will be interrupted if the '`--all`' option is specified. If the '`--thread-group`' option is specified, all threads in that group will be interrupted.

GDB Command

The corresponding GDB command is '`interrupt`'.

Example

```
(gdb)
111-exec-continue
111^running

(gdb)
222-exec-interrupt
222^done
(gdb)
111*stopped,signal-name="SIGINT",signal-meaning="Interrupt",
frame={addr="0x00010140",func="foo",args=[],file="try.c",
fullname="/home/foo/bar/try.c",line="13",arch="i386:x86_64"}
(gdb)

(gdb)
-exec-interrupt
^error,msg="mi_cmd_exec_interrupt: Inferior not executing."
(gdb)
```

The -exec-jump Command

Synopsis

```
-exec-jump location
```

Resumes execution of the inferior program at the location specified by parameter. See Section 9.2 [Specify Location], page 120, for a description of the different forms of *location*.

GDB Command

The corresponding GDB command is '`jump`'.

```
addr="0x000100d4",line="5",file="hello.c"
(gdb)
```

The -exec-return Command

Synopsis

```
-exec-return
```

Makes current function return immediately. Doesn't execute the inferior. Displays the new current frame.

GDB Command

The corresponding GDB command is '`return`'.

Example

```
(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",disp="keep",bkptno="1",
frame={func="callee4",args=[],
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="8",
arch="i386:x86_64"}
(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",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="18",
arch="i386:x86_64"}
(gdb)
```

The -exec-run Command

Synopsis

```
-exec-run [ --all | --thread-group N ] [ --start ]
```

Starts execution of the inferior from the beginning. The inferior executes until either a breakpoint is encountered or the program exits. In the latter case the output will include an exit code, if the program has exited exceptionally.

When neither the '`--all`' nor the '`--thread-group`' option is specified, the current inferior is started. If the '`--thread-group`' option is specified, it should refer to a thread group of type '`process`', and that thread group will be started. If the '`--all`' option is specified, then all inferiors will be started.

Using the '`--start`' option instructs the debugger to stop the execution at the start of the inferior's main subprogram, following the same behavior as the `start` command (see Section 4.2 [Starting], page 32).

GDB Command

The corresponding GDB command is '`run`'.

Examples

```
(gdb)
-break-insert main
^done,bkpt={number="1",addr="0x0001072c",file="recursive2.c",line="4"}
(gdb)
-exec-run
^running
(gdb)
*stopped,reason="breakpoint-hit",disp="keep",bkptno="1",
frame={func="main",args=[],file="recursive2.c",
fullname="/home/foo/bar/recursive2.c",line="4",arch="i386:x86_64"}
(gdb)
```

Program exited normally:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited-normally"
(gdb)
```

Program exited exceptionally:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited",exit-code="01"
(gdb)
```

Another way the program can terminate is if it receives a signal such as `SIGINT`. In this case, GDB/MI displays this:

```
(gdb)
*stopped,reason="exited-signalled",signal-name="SIGINT",
signal-meaning="Interrupt"
```

The `-exec-step` Command

Synopsis

```
-exec-step [--reverse]
```

Resumes execution of the inferior program, stopping when the beginning of the next source line is reached, if the next source line is not a function call. If it is, stop at the first instruction of the called function. If the '`--reverse`' option is specified, resumes reverse execution of the inferior program, stopping at the beginning of the previously executed source line.

GDB Command

The corresponding GDB command is '`step`'.

Example

Stepping into a function:

```
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",
frame={func="foo",args=[{name="a",value="10"},
{name="b",value="0"}],file="recursive2.c",
fullname="/home/foo/bar/recursive2.c",line="11",arch="i386:x86_64"}
(gdb)
```

Regular stepping:

```
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",line="14",file="recursive2.c"
(gdb)
```

The `-exec-step-instruction` Command

Synopsis

```
-exec-step-instruction [--reverse]
```

Resumes the inferior which executes one machine instruction. If the '`--reverse`' option is specified, resumes reverse execution of the inferior program, stopping at the previously executed instruction. The output, once GDB has stopped, will vary depending on whether we have stopped in the middle of a source line or not. In the former case, the address at which the program stopped will be printed as well.

GDB Command

The corresponding GDB command is '`stepi`'.

Example

```
(gdb)
-exec-step-instruction
^running

(gdb)
*stopped,reason="end-stepping-range",
frame={func="foo",args=[],file="try.c",
fullname="/home/foo/bar/try.c",line="10",arch="i386:x86_64"}
(gdb)
-exec-step-instruction
^running

(gdb)
*stopped,reason="end-stepping-range",
frame={addr="0x000100f4",func="foo",args=[],file="try.c",
fullname="/home/foo/bar/try.c",line="10",arch="i386:x86_64"}
(gdb)
```

The `-exec-until` Command

Synopsis

```
-exec-until [ location ]
```

Executes the inferior until the *location* specified in the argument is reached. If there is no argument, the inferior executes until a source line greater than the current one is reached. The reason for stopping in this case will be '`location-reached`'.

GDB Command

The corresponding GDB command is '`until`'.

Example

```
(gdb)
-exec-until recursive2.c:6
^running
(gdb)
x = 55
*stopped,reason="location-reached",frame={func="main",args=[],
file="recursive2.c",fullname="/home/foo/bar/recursive2.c",line="6",
arch="i386:x86_64"}
(gdb)
```

27.16 GDB/MI Stack Manipulation Commands

The `-enable-frame-filters` Command

```
-enable-frame-filters
```

GDB allows Python-based frame filters to affect the output of the MI commands relating to stack traces. As there is no way to implement this in a fully backward-compatible way, a front end must request that this functionality be enabled.

Once enabled, this feature cannot be disabled.

Note that if Python support has not been compiled into GDB, this command will still succeed (and do nothing).

The `-stack-info-frame` Command

Synopsis

```
-stack-info-frame
```

Get info on the selected frame.

GDB Command

The corresponding GDB command is '`info frame`' or '`frame`' (without arguments).

Example

```
(gdb)
-stack-info-frame
^done,frame={level="1",addr="0x0001076c",func="callee3",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="17",
```

```
arch="i386:x86_64"}
(gdb)
```

The -stack-info-depth Command

Synopsis

```
-stack-info-depth [ max-depth ]
```

Return the depth of the stack. If the integer argument *max-depth* is specified, do not count beyond *max-depth* frames.

GDB Command

There's no equivalent GDB command.

Example

For a stack with frame levels 0 through 11:

```
(gdb)
-stack-info-depth
^done,depth="12"
(gdb)
-stack-info-depth 4
^done,depth="4"
(gdb)
-stack-info-depth 12
^done,depth="12"
(gdb)
-stack-info-depth 11
^done,depth="11"
(gdb)
-stack-info-depth 13
^done,depth="12"
(gdb)
```

The -stack-list-arguments Command

Synopsis

```
-stack-list-arguments [ --no-frame-filters ] [ --skip-unavailable ] print-values
   [ low-frame high-frame ]
```

Display a list of the arguments for the frames between *low-frame* and *high-frame* (inclusive). If *low-frame* and *high-frame* are not provided, list the arguments for the whole call stack. If the two arguments are equal, show the single frame at the corresponding level. It is an error if *low-frame* is larger than the actual number of frames. On the other hand, *high-frame* may be larger than the actual number of frames, in which case only existing frames will be returned.

If *print-values* is 0 or `--no-values`, print only the names of the variables; if it is 1 or `--all-values`, print also their values; and if it is 2 or `--simple-values`, print the name, type and value for simple data types, and the name and type for arrays, structures and unions. If the option `--no-frame-filters` is supplied, then Python frame filters will not be executed.

If the `--skip-unavailable` option is specified, arguments that are not available are not listed. Partially available arguments are still displayed, however.

Use of this command to obtain arguments in a single frame is deprecated in favor of the '`-stack-list-variables`' command.

GDB Command

GDB does not have an equivalent command. `gdbtk` has a '`gdb_get_args`' command which partially overlaps with the functionality of '`-stack-list-arguments`'.

Example

```
(gdb)
-stack-list-frames
^done,
stack=[
frame={level="0",addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="8",
arch="i386:x86_64"},
frame={level="1",addr="0x0001076c",func="callee3",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="17",
arch="i386:x86_64"},
frame={level="2",addr="0x0001078c",func="callee2",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="22",
arch="i386:x86_64"},
frame={level="3",addr="0x000107b4",func="callee1",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="27",
arch="i386:x86_64"},
frame={level="4",addr="0x000107e0",func="main",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",
fullname="/home/foo/bar/devo/gdb/testsuite/gdb.mi/basics.c",line="32",
arch="i386:x86_64"}]
(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)
```

Synopsis

```
-stack-select-frame framenum
```

Change the selected frame. Select a different frame *framenum* on the stack.

This command in deprecated in favor of passing the '`--frame`' option to every command.

GDB Command

The corresponding GDB commands are '`frame`', '`up`', '`down`', '`select-frame`', '`up-silent`', and '`down-silent`'.

Example

```
(gdb)
-stack-select-frame 2
^done
(gdb)
```

27.17 GDB/MI Variable Objects

Introduction to Variable Objects

Variable objects are "object-oriented" MI interface for examining and changing values of expressions. Unlike some other MI interfaces that work with expressions, variable objects are specifically designed for simple and efficient presentation in the frontend. A variable object is identified by string name. When a variable object is created, the frontend specifies the expression for that variable object. The expression can be a simple variable, or it can be an arbitrary complex expression, and can even involve CPU registers. After creating a variable object, the frontend can invoke other variable object operations—for example to obtain or change the value of a variable object, or to change display format.

Variable objects have hierarchical tree structure. Any variable object that corresponds to a composite type, such as structure in C, has a number of child variable objects, for example corresponding to each element of a structure. A child variable object can itself have children, recursively. Recursion ends when we reach leaf variable objects, which always have built-in types. Child variable objects are created only by explicit request, so if a frontend is not interested in the children of a particular variable object, no child will be created.

For a leaf variable object it is possible to obtain its value as a string, or set the value from a string. String value can be also obtained for a non-leaf variable object, but it's generally a string that only indicates the type of the object, and does not list its contents. Assignment to a non-leaf variable object is not allowed.

A frontend does not need to read the values of all variable objects each time the program stops. Instead, MI provides an update command that lists all variable objects whose values has changed since the last update operation. This considerably reduces the amount of data that must be transferred to the frontend. As noted above, children variable objects are created on demand, and only leaf variable objects have a real value. As result, gdb will read target memory only for leaf variables that frontend has created.

The automatic update is not always desirable. For example, a frontend might want to keep a value of some expression for future reference, and never update it. For another example, fetching memory is relatively slow for embedded targets, so a frontend might want

to disable automatic update for the variables that are either not visible on the screen, or "closed". This is possible using so called "frozen variable objects". Such variable objects are never implicitly updated.

Variable objects can be either *fixed* or *floating*. For the fixed variable object, the expression is parsed when the variable object is created, including associating identifiers to specific variables. The meaning of expression never changes. For a floating variable object the values of variables whose names appear in the expressions are re-evaluated every time in the context of the current frame. Consider this example:

```
void do_work(...)
{
        struct work_state state;

        if (...)
           do_work(...);
}
```

If a fixed variable object for the `state` variable is created in this function, and we enter the recursive call, the variable object will report the value of `state` in the top-level `do_work` invocation. On the other hand, a floating variable object will report the value of `state` in the current frame.

If an expression specified when creating a fixed variable object refers to a local variable, the variable object becomes bound to the thread and frame in which the variable object is created. When such variable object is updated, GDB makes sure that the thread/frame combination the variable object is bound to still exists, and re-evaluates the variable object in context of that thread/frame.

The following is the complete set of GDB/MI operations defined to access this functionality:

Operation	**Description**
`-enable-pretty-printing`	enable Python-based pretty-printing
`-var-create`	create a variable object
`-var-delete`	delete the variable object and/or its children
`-var-set-format`	set the display format of this variable
`-var-show-format`	show the display format of this variable
`-var-info-num-children`	tells how many children this object has
`-var-list-children`	return a list of the object's children
`-var-info-type`	show the type of this variable object
`-var-info-expression`	print parent-relative expression that this variable object represents
`-var-info-path-expression`	print full expression that this variable object represents
`-var-show-attributes`	is this variable editable? does it exist here?
`-var-evaluate-expression`	get the value of this variable
`-var-assign`	set the value of this variable
`-var-update`	update the variable and its children
`-var-set-frozen`	set frozenness attribute
`-var-set-update-range`	set range of children to display on update

The zero-hexadecimal format has a representation similar to hexadecimal but with padding zeroes to the left of the value. For example, a 32-bit hexadecimal value of 0x1234 would be represented as 0x00001234 in the zero-hexadecimal format.

For a variable with children, the format is set only on the variable itself, and the children are not affected.

The `-var-show-format` Command

Synopsis

```
-var-show-format name
```

Returns the format used to display the value of the object *name*.

```
format ↦
format-spec
```

The `-var-info-num-children` Command

Synopsis

```
-var-info-num-children name
```

Returns the number of children of a variable object *name*:

```
numchild=n
```

Note that this number is not completely reliable for a dynamic varobj. It will return the current number of children, but more children may be available.

The `-var-list-children` Command

Synopsis

```
-var-list-children [print-values] name [from to]
```

Return a list of the children of the specified variable object and create variable objects for them, if they do not already exist. With a single argument or if *print-values* has a value of 0 or `--no-values`, print only the names of the variables; if *print-values* is 1 or `--all-values`, also print their values; and if it is 2 or `--simple-values` print the name and value for simple data types and just the name for arrays, structures and unions.

from and *to*, if specified, indicate the range of children to report. If *from* or *to* is less than zero, the range is reset and all children will be reported. Otherwise, children starting at *from* (zero-based) and up to and excluding *to* will be reported.

If a child range is requested, it will only affect the current call to `-var-list-children`, but not future calls to `-var-update`. For this, you must instead use `-var-set-update-range`. The intent of this approach is to enable a front end to implement any update approach it likes; for example, scrolling a view may cause the front end to request more children with `-var-list-children`, and then the front end could call `-var-set-update-range` with a different range to ensure that future updates are restricted to just the visible items.

For each child the following results are returned:

name Name of the variable object created for this child.

exp The expression to be shown to the user by the front end to designate this child. For example this may be the name of a structure member.

For a dynamic varobj, this value cannot be used to form an expression. There is no way to do this at all with a dynamic varobj.

For C/C++ structures there are several pseudo children returned to designate access qualifiers. For these pseudo children *exp* is '`public`', '`private`', or '`protected`'. In this case the type and value are not present.

A dynamic varobj will not report the access qualifying pseudo-children, regardless of the language. This information is not available at all with a dynamic varobj.

numchild Number of children this child has. For a dynamic varobj, this will be 0.

type The type of the child. If '`print object`' (see Section 10.9 [Print Settings], page 149) is set to `on`, the *actual* (derived) type of the object is shown rather than the *declared* one.

value If values were requested, this is the value.

thread-id If this variable object is associated with a thread, this is the thread's global thread id. Otherwise this result is not present.

frozen If the variable object is frozen, this variable will be present with a value of 1.

displayhint

A dynamic varobj can supply a display hint to the front end. The value comes directly from the Python pretty-printer object's `display_hint` method. See Section 23.3.2.5 [Pretty Printing API], page 397.

dynamic This attribute will be present and have the value '`1`' if the varobj is a dynamic varobj. If the varobj is not a dynamic varobj, then this attribute will not be present.

The result may have its own attributes:

'`displayhint`'

A dynamic varobj can supply a display hint to the front end. The value comes directly from the Python pretty-printer object's `display_hint` method. See Section 23.3.2.5 [Pretty Printing API], page 397.

'`has_more`'

This is an integer attribute which is nonzero if there are children remaining after the end of the selected range.

Example

```
(gdb)
 -var-list-children n
 ^done,numchild=n,children=[child={name=name,exp=exp,
 numchild=n,type=type},(repeats N times)]
(gdb)
 -var-list-children --all-values n
 ^done,numchild=n,children=[child={name=name,exp=exp,
 numchild=n,value=value,type=type},(repeats N times)]
```

The -var-info-type Command

Synopsis

```
-var-info-type name
```

Returns the type of the specified variable *name*. The type is returned as a string in the same format as it is output by the GDB CLI:

```
type=typename
```

The -var-info-expression Command

Synopsis

```
-var-info-expression name
```

Returns a string that is suitable for presenting this variable object in user interface. The string is generally not valid expression in the current language, and cannot be evaluated.

For example, if `a` is an array, and variable object `A` was created for `a`, then we'll get this output:

```
(gdb) -var-info-expression A.1
^done,lang="C",exp="1"
```

Here, the value of `lang` is the language name, which can be found in Section 15.4 [Supported Languages], page 221.

Note that the output of the `-var-list-children` command also includes those expressions, so the `-var-info-expression` command is of limited use.

The -var-info-path-expression Command

Synopsis

```
-var-info-path-expression name
```

Returns an expression that can be evaluated in the current context and will yield the same value that a variable object has. Compare this with the `-var-info-expression` command, which result can be used only for UI presentation. Typical use of the `-var-info-path-expression` command is creating a watchpoint from a variable object.

This command is currently not valid for children of a dynamic varobj, and will give an error when invoked on one.

For example, suppose `C` is a C++ class, derived from class `Base`, and that the `Base` class has a member called `m_size`. Assume a variable `c` is has the type of `C` and a variable object `C` was created for variable `c`. Then, we'll get this output:

```
(gdb) -var-info-path-expression C.Base.public.m_size
^done,path_expr=((Base)c).m_size)
```

The -var-show-attributes Command

Synopsis

```
-var-show-attributes name
```

List attributes of the specified variable object *name*:

```
status=attr [ ( ,attr )* ]
```

where *attr* is `{ { editable | noneditable } | TBD }`.

The -var-evaluate-expression Command

Synopsis

```
-var-evaluate-expression [-f format-spec] name
```

Evaluates the expression that is represented by the specified variable object and returns its value as a string. The format of the string can be specified with the '-f' option. The possible values of this option are the same as for -var-set-format (see [-var-set-format], page 590). If the '-f' option is not specified, the current display format will be used. The current display format can be changed using the -var-set-format command.

```
value=value
```

Note that one must invoke -var-list-children for a variable before the value of a child variable can be evaluated.

The -var-assign Command

Synopsis

```
-var-assign name expression
```

Assigns the value of *expression* to the variable object specified by *name*. The object must be 'editable'. If the variable's value is altered by the assign, the variable will show up in any subsequent -var-update list.

Example

```
(gdb)
-var-assign var1 3
^done,value="3"
(gdb)
-var-update *
^done,changelist=[{name="var1",in_scope="true",type_changed="false"}]
(gdb)
```

The -var-update Command

Synopsis

```
-var-update [print-values] {name | "*"}
```

Reevaluate the expressions corresponding to the variable object *name* and all its direct and indirect children, and return the list of variable objects whose values have changed; *name* must be a root variable object. Here, "changed" means that the result of -var-evaluate-expression before and after the -var-update is different. If '*' is used as the variable object names, all existing variable objects are updated, except for frozen ones (see [-var-set-frozen], page 596). The option *print-values* determines whether both names and values, or just names are printed. The possible values of this option are the same as for -var-list-children (see [-var-list-children], page 591). It is recommended to use the '--all-values' option, to reduce the number of MI commands needed on each program stop.

With the '*' parameter, if a variable object is bound to a currently running thread, it will not be updated, without any diagnostic.

If -var-set-update-range was previously used on a varobj, then only the selected range of children will be reported.

'`has_more`'
: This is an integer value, which will be 1 if there are more children available outside the varobj's update range.

'`dynamic`'
: This attribute will be present and have the value '`1`' if the varobj is a dynamic varobj. If the varobj is not a dynamic varobj, then this attribute will not be present.

'`new_children`'
: If new children were added to a dynamic varobj within the selected update range (as set by `-var-set-update-range`), then they will be listed in this attribute.

Example

```
(gdb)
-var-assign var1 3
^done,value="3"
(gdb)
-var-update --all-values var1
^done,changelist=[{name="var1",value="3",in_scope="true",
type_changed="false"}]
(gdb)
```

The `-var-set-frozen` Command

Synopsis

```
-var-set-frozen name flag
```

Set the frozenness flag on the variable object *name*. The *flag* parameter should be either '`1`' to make the variable frozen or '`0`' to make it unfrozen. If a variable object is frozen, then neither itself, nor any of its children, are implicitly updated by `-var-update` of a parent variable or by `-var-update *`. Only `-var-update` of the variable itself will update its value and values of its children. After a variable object is unfrozen, it is implicitly updated by all subsequent `-var-update` operations. Unfreezing a variable does not update it, only subsequent `-var-update` does.

Example

```
(gdb)
-var-set-frozen V 1
^done
(gdb)
```

The `-var-set-update-range` command

Synopsis

```
-var-set-update-range name from to
```

Set the range of children to be returned by future invocations of `-var-update`.

from and *to* indicate the range of children to report. If *from* or *to* is less than zero, the range is reset and all children will be reported. Otherwise, children starting at *from* (zero-based) and up to and excluding *to* will be reported.

Example

```
(gdb)
-var-set-update-range V 1 2
^done
```

The `-var-set-visualizer` command

Synopsis

```
-var-set-visualizer name visualizer
```

Set a visualizer for the variable object *name*.

visualizer is the visualizer to use. The special value '`None`' means to disable any visualizer in use.

If not '`None`', *visualizer* must be a Python expression. This expression must evaluate to a callable object which accepts a single argument. GDB will call this object with the value of the varobj *name* as an argument (this is done so that the same Python pretty-printing code can be used for both the CLI and MI). When called, this object must return an object which conforms to the pretty-printing interface (see Section 23.3.2.5 [Pretty Printing API], page 397).

The pre-defined function `gdb.default_visualizer` may be used to select a visualizer by following the built-in process (see Section 23.3.2.6 [Selecting Pretty-Printers], page 398). This is done automatically when a varobj is created, and so ordinarily is not needed.

This feature is only available if Python support is enabled. The MI command `-list-features` (see Section 27.25 [GDB/MI Support Commands], page 630) can be used to check this.

Example

Resetting the visualizer:

```
(gdb)
-var-set-visualizer V None
^done
```

Reselecting the default (type-based) visualizer:

```
(gdb)
-var-set-visualizer V gdb.default_visualizer
^done
```

Suppose `SomeClass` is a visualizer class. A lambda expression can be used to instantiate this class for a varobj:

```
(gdb)
-var-set-visualizer V "lambda val: SomeClass()"
^done
```

27.18 GDB/MI Data Manipulation

This section describes the GDB/MI commands that manipulate data: examine memory and registers, evaluate expressions, etc.

For details about what an addressable memory unit is, see [addressable memory unit], page 146.

`func-name`
: The name of the function this instruction is within.

`offset`
: The decimal offset in bytes from the start of '`func-name`'.

`inst`
: The text disassembly for this '`address`'.

`opcodes`
: This field is only present for modes 2, 3 and 5. This contains the raw opcode bytes for the '`inst`' field.

For modes 1, 3, 4 and 5 the '`asm_insns`' list contains tuples named '`src_and_asm_line`', each of which has the following fields:

`line`
: The line number within '`file`'.

`file`
: The file name from the compilation unit. This might be an absolute file name or a relative file name depending on the compile command used.

`fullname`
: Absolute file name of '`file`'. It is converted to a canonical form using the source file search path (see Section 9.5 [Specifying Source Directories], page 124) and after resolving all the symbolic links.

  If the source file is not found this field will contain the path as present in the debug information.

`line_asm_insn`
: This is a list of tuples containing the disassembly for '`line`' in '`file`'. The fields of each tuple are the same as for `-data-disassemble` in *mode* 0 and 2, so '`address`', '`func-name`', '`offset`', '`inst`', and optionally '`opcodes`'.

Note that whatever included in the '`inst`' field, is not manipulated directly by GDB/MI, i.e., it is not possible to adjust its format.

GDB Command

The corresponding GDB command is '`disassemble`'.

Example

Disassemble from the current value of `$pc` to `$pc + 20`:

```
(gdb)
-data-disassemble -s $pc -e "$pc + 20" -- 0
^done,
asm_insns=[
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"},
{address="0x000107c8",func-name="main",offset="12",
inst="or  %o2, 0x140, %o1\t! 0x11940 <_lib_version+8>"},
{address="0x000107cc",func-name="main",offset="16",
inst="sethi  %hi(0x11800), %o2"},
{address="0x000107d0",func-name="main",offset="20",
inst="or  %o2, 0x168, %o4\t! 0x11968 <_lib_version+48>"}]
(gdb)
```

Disassemble the whole `main` function. Line 32 is part of `main`.

```
-data-disassemble -f basics.c -l 32 -- 0
```

```
^done,asm_insns=[
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"},
{address="0x000107c0",func-name="main",offset="4",
inst="mov   2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi %hi(0x11800), %o2"},
[...]
{address="0x0001081c",func-name="main",offset="96",inst="ret "},
{address="0x00010820",func-name="main",offset="100",inst="restore "}]
(gdb)
```

Disassemble 3 instructions from the start of `main`:

```
(gdb)
-data-disassemble -f basics.c -l 32 -n 3 -- 0
^done,asm_insns=[
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"},
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"}]
(gdb)
```

Disassemble 3 instructions from the start of `main` in mixed mode:

```
(gdb)
-data-disassemble -f basics.c -l 32 -n 3 -- 1
^done,asm_insns=[
src_and_asm_line={line="31",
file="../../../src/gdb/testsuite/gdb.mi/basics.c",
fullname="/absolute/path/to/src/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="../../../src/gdb/testsuite/gdb.mi/basics.c",
fullname="/absolute/path/to/src/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)
```

The `-data-evaluate-expression` Command

Synopsis

```
-data-evaluate-expression expr
```

Evaluate *expr* as an expression. The expression could contain an inferior function call. The function call will execute synchronously. If the expression contains spaces, it must be enclosed in double quotes.

GDB Command

The corresponding GDB commands are '`print`', '`output`', and '`call`'. In `gdbtk` only, there's a corresponding '`gdb_eval`' command.

Example

In the following example, the numbers that precede the commands are the *tokens* described in Section 27.4 [GDB/MI Command Syntax], page 540. Notice how GDB/MI returns the same tokens in its output.

```
211-data-evaluate-expression A
211^done,value="1"
(gdb)
311-data-evaluate-expression &A
311^done,value="0xefffeb7c"
(gdb)
411-data-evaluate-expression A+3
411^done,value="4"
(gdb)
511-data-evaluate-expression "A + 3"
511^done,value="4"
(gdb)
```

The `-data-list-changed-registers` Command

Synopsis

```
-data-list-changed-registers
```

Display a list of the registers that have changed.

GDB Command

GDB doesn't have a direct analog for this command; `gdbtk` has the corresponding command '`gdb_changed_register_list`'.

Example

On a PPC MBX board:

```
(gdb)
-exec-continue
^running

(gdb)
*stopped,reason="breakpoint-hit",disp="keep",bkptno="1",frame={
func="main",args=[],file="try.c",fullname="/home/foo/bar/try.c",
line="5",arch="powerpc"}
(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)
```

The `-data-list-register-names` Command

Synopsis

```
-data-list-register-names [ ( regno )+ ]
```

Show a list of register names for the current target. If no arguments are given, it shows a list of the names of all the registers. If integer numbers are given as arguments, it will print a list of the names of the registers corresponding to the arguments. To ensure consistency between a register name and its number, the output list may include empty register names.

GDB Command

GDB does not have a command which corresponds to '`-data-list-register-names`'. In `gdbtk` there is a corresponding command '`gdb_regnames`'.

Example

For the PPC MBX board:

```
(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)
```

The `-data-list-register-values` Command

Synopsis

```
-data-list-register-values
   [ --skip-unavailable ] fmt [ ( regno )*]
```

Display the registers' contents. The format according to which the registers' contents are to be returned is given by *fmt*, followed by an optional list of numbers specifying the registers to display. A missing list of numbers indicates that the contents of all the registers must be returned. The `--skip-unavailable` option indicates that only the available registers are to be returned.

Allowed formats for *fmt* are:

`x`	Hexadecimal
`o`	Octal
`t`	Binary
`d`	Decimal
`r`	Raw
`N`	Natural

GDB Command

The corresponding GDB commands are '`info reg`', '`info all-reg`', and (in `gdbtk`) '`gdb_fetch_registers`'.

Example

For a PPC MBX board (note: line breaks are for readability only, they don't appear in the actual output):

```
(gdb)
```

```
-data-list-register-values r 64 65
^done,register-values=[{number="64",value="0xfe00a300"},
{number="65",value="0x00029002"}]
(gdb)
-data-list-register-values x
^done,register-values=[{number="0",value="0xfe0043c8"},
{number="1",value="0x3fff88"},{number="2",value="0xfffffffe"},
{number="3",value="0x0"},{number="4",value="0xa"},
{number="5",value="0x3fff68"},{number="6",value="0x3fff58"},
{number="7",value="0xfe011e98"},{number="8",value="0x2"},
{number="9",value="0xfa202820"},{number="10",value="0xfa202808"},
{number="11",value="0x1"},{number="12",value="0x0"},
{number="13",value="0x4544"},{number="14",value="0xffdfffff"},
{number="15",value="0xffffffff"},{number="16",value="0xfffffeff"},
{number="17",value="0xefffffed"},{number="18",value="0xfffffffe"},
{number="19",value="0xffffffff"},{number="20",value="0xffffffff"},
{number="21",value="0xffffffff"},{number="22",value="0xfffffff7"},
{number="23",value="0xffffffff"},{number="24",value="0xffffffff"},
{number="25",value="0xffffffff"},{number="26",value="0xfffffffb"},
{number="27",value="0xffffffff"},{number="28",value="0xf7bfffff"},
{number="29",value="0x0"},{number="30",value="0xfe010000"},
{number="31",value="0x0"},{number="32",value="0x0"},
{number="33",value="0x0"},{number="34",value="0x0"},
{number="35",value="0x0"},{number="36",value="0x0"},
{number="37",value="0x0"},{number="38",value="0x0"},
{number="39",value="0x0"},{number="40",value="0x0"},
{number="41",value="0x0"},{number="42",value="0x0"},
{number="43",value="0x0"},{number="44",value="0x0"},
{number="45",value="0x0"},{number="46",value="0x0"},
{number="47",value="0x0"},{number="48",value="0x0"},
{number="49",value="0x0"},{number="50",value="0x0"},
{number="51",value="0x0"},{number="52",value="0x0"},
{number="53",value="0x0"},{number="54",value="0x0"},
{number="55",value="0x0"},{number="56",value="0x0"},
{number="57",value="0x0"},{number="58",value="0x0"},
{number="59",value="0x0"},{number="60",value="0x0"},
{number="61",value="0x0"},{number="62",value="0x0"},
{number="63",value="0x0"},{number="64",value="0xfe00a300"},
{number="65",value="0x29002"},{number="66",value="0x202f04b5"},
{number="67",value="0xfe0043b0"},{number="68",value="0xfe00b3e4"},
{number="69",value="0x20002b03"}]
(gdb)
```

The `-data-read-memory` Command

This command is deprecated, use `-data-read-memory-bytes` instead.

Synopsis

```
-data-read-memory [ -o byte-offset ]
  address word-format word-size
  nr-rows nr-cols [ aschar ]
```

where:

‘*address*’ An expression specifying the address of the first memory word to be read. Complex expressions containing embedded white space should be quoted using the C convention.

'*word-format*'
: The format to be used to print the memory words. The notation is the same as for GDB's `print` command (see Section 10.5 [Output Formats], page 142).

'*word-size*'
: The size of each memory word in bytes.

'*nr-rows*' The number of rows in the output table.

'*nr-cols*' The number of columns in the output table.

'*aschar*' If present, indicates that each row should include an ASCII dump. The value of *aschar* is used as a padding character when a byte is not a member of the printable ASCII character set (printable ASCII characters are those whose code is between 32 and 126, inclusively).

'*byte-offset*'
: An offset to add to the *address* before fetching memory.

This command displays memory contents as a table of *nr-rows* by *nr-cols* words, each word being *word-size* bytes. In total, *nr-rows* * *nr-cols* * *word-size* bytes are read (returned as '`total-bytes`'). Should less than the requested number of bytes be returned by the target, the missing words are identified using '`N/A`'. The number of bytes read from the target is returned in '`nr-bytes`' and the starting address used to read memory in '`addr`'.

The address of the next/previous row or page is available in '`next-row`' and '`prev-row`', '`next-page`' and '`prev-page`'.

GDB Command

The corresponding GDB command is '`x`'. `gdbtk` has '`gdb_get_mem`' memory read command.

Example

Read six bytes of memory starting at `bytes+6` but then offset by `-6` bytes. Format as three rows of two columns. One byte per word. Display each word in hex.

```
(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)
```

Read two bytes of memory starting at address `shorts + 64` and display as a single word formatted in decimal.

```
(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)
```

Read thirty two bytes of memory starting at `bytes+16` and format as eight rows of four columns. Include a string encoding with '`x`' used as the non-printable character.

```
(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)
```

The `-data-read-memory-bytes` Command

Synopsis

```
-data-read-memory-bytes [ -o offset ]
  address count
```

where:

'*address*' An expression specifying the address of the first addressable memory unit to be read. Complex expressions containing embedded white space should be quoted using the C convention.

'*count*' The number of addressable memory units to read. This should be an integer literal.

'*offset*' The offset relative to *address* at which to start reading. This should be an integer literal. This option is provided so that a frontend is not required to first evaluate address and then perform address arithmetics itself.

This command attempts to read all accessible memory regions in the specified range. First, all regions marked as unreadable in the memory map (if one is defined) will be skipped. See Section 10.18 [Memory Region Attributes], page 173. Second, GDB will attempt to read the remaining regions. For each one, if reading full region results in an errors, GDB will try to read a subset of the region.

In general, every single memory unit in the region may be readable or not, and the only way to read every readable unit is to try a read at every address, which is not practical. Therefore, GDB will attempt to read all accessible memory units at either beginning or the end of the region, using a binary division scheme. This heuristic works well for reading across a memory map boundary. Note that if a region has a readable range that is neither at the beginning or the end, GDB will not read it.

The result record (see Section 27.7.1 [GDB/MI Result Records], page 544) that is output of the command includes a field named '`memory`' whose content is a list of tuples. Each tuple represent a successfully read memory block and has the following fields:

`begin` The start address of the memory block, as hexadecimal literal.

`end` The end address of the memory block, as hexadecimal literal.

`offset` The offset of the memory block, as hexadecimal literal, relative to the start address passed to `-data-read-memory-bytes`.

`contents` The contents of the memory block, in hex.

GDB Command

The corresponding GDB command is '`x`'.

Example

```
(gdb)
-data-read-memory-bytes &a 10
^done,memory=[{begin="0xbffff154",offset="0x00000000",
               end="0xbffff15e",
               contents="01000000020000000300"}]
(gdb)
```

The `-data-write-memory-bytes` Command

Synopsis

```
 -data-write-memory-bytes address contents
 -data-write-memory-bytes address contents [count]
```

where:

'*address*' An expression specifying the address of the first addressable memory unit to be written. Complex expressions containing embedded white space should be quoted using the C convention.

'*contents*'
The hex-encoded data to write. It is an error if *contents* does not represent an integral number of addressable memory units.

'*count*' Optional argument indicating the number of addressable memory units to be written. If *count* is greater than *contents*' length, GDB will repeatedly write *contents* until it fills *count* memory units.

GDB Command

There's no corresponding GDB command.

Example

```
(gdb)
-data-write-memory-bytes &a "aabbccdd"
^done
(gdb)
(gdb)
-data-write-memory-bytes &a "aabbccdd" 16e
^done
(gdb)
```

27.19 GDB/MI Tracepoint Commands

The commands defined in this section implement MI support for tracepoints. For detailed introduction, see Chapter 13 [Tracepoints], page 193.

The `-trace-find` Command

Synopsis

```
-trace-find mode [parameters...]
```

Find a trace frame using criteria defined by *mode* and *parameters*. The following table lists permissible modes and their parameters. For details of operation, see Section 13.2.1 [tfind], page 205.

'`none`' No parameters are required. Stops examining trace frames.

'`frame-number`'
An integer is required as parameter. Selects tracepoint frame with that index.

'`tracepoint-number`'
An integer is required as parameter. Finds next trace frame that corresponds to tracepoint with the specified number.

'`pc`' An address is required as parameter. Finds next trace frame that corresponds to any tracepoint at the specified address.

'`pc-inside-range`'
Two addresses are required as parameters. Finds next trace frame that corresponds to a tracepoint at an address inside the specified range. Both bounds are considered to be inside the range.

'`pc-outside-range`'
Two addresses are required as parameters. Finds next trace frame that corresponds to a tracepoint at an address outside the specified range. Both bounds are considered to be inside the range.

'`line`' Line specification is required as parameter. See Section 9.2 [Specify Location], page 120. Finds next trace frame that corresponds to a tracepoint at the specified location.

If '`none`' was passed as *mode*, the response does not have fields. Otherwise, the response may have the following fields:

'`found`' This field has either '`0`' or '`1`' as the value, depending on whether a matching tracepoint was found.

'`traceframe`'
The index of the found traceframe. This field is present iff the '`found`' field has value of '`1`'.

'`tracepoint`'
The index of the found tracepoint. This field is present iff the '`found`' field has value of '`1`'.

'`frame`' The information about the frame corresponding to the found trace frame. This field is present only if a trace frame was found. See Section 27.7.5 [GDB/MI Frame Information], page 552, for description of this field.

GDB Command

The corresponding GDB command is '`tfind`'.

```
      memory=[{address="0x0000000000602264",length="4"},
              {address="0x0000000000615bc0",length="4"}]
    (gdb)
```

Where:

`explicit-variables`
: The set of objects that have been collected in their entirety (as opposed to collecting just a few elements of an array or a few struct members). For each object, its name and value are printed. The `--var-print-values` option affects how or whether the value field is output. If *var_pval* is 0, then print only the names; if it is 1, print also their values; and if it is 2, print the name, type and value for simple data types, and the name and type for arrays, structures and unions.

`computed-expressions`
: The set of computed expressions that have been collected at the current trace frame. The `--comp-print-values` option affects this set like the `--var-print-values` option affects the `explicit-variables` set. See above.

`registers`
: The registers that have been collected at the current trace frame. For each register collected, the name and current value are returned. The value is formatted according to the `--registers-format` option. See the `-data-list-register-values` command for a list of the allowed formats. The default is '`x`'.

`tvars`
: The trace state variables that have been collected at the current trace frame. For each trace state variable collected, the name and current value are returned.

`memory`
: The set of memory ranges that have been collected at the current trace frame. Its content is a list of tuples. Each tuple represents a collected memory range and has the following fields:

  `address`
  : The start address of the memory range, as hexadecimal literal.

  `length`
  : The length of the memory range, as decimal literal.

  `contents`
  : The contents of the memory block, in hex. This field is only present if the `--memory-contents` option is specified.

GDB Command

There is no corresponding GDB command.

Example

-trace-list-variables

Synopsis

```
-trace-list-variables
```

Return a table of all defined trace variables. Each element of the table has the following fields:

'`name`'
: The name of the trace variable. This field is always present.

‘`initial`’ The initial value. This is a 64-bit signed integer. This field is always present.

‘`current`’ The value the trace variable has at the moment. This is a 64-bit signed integer. This field is absent iff current value is not defined, for example if the trace was never run, or is presently running.

GDB Command

The corresponding GDB command is ‘`tvariables`’.

Example

```
(gdb)
-trace-list-variables
^done,trace-variables={nr_rows="1",nr_cols="3",
hdr=[{width="15",alignment="-1",col_name="name",colhdr="Name"},
     {width="11",alignment="-1",col_name="initial",colhdr="Initial"},
     {width="11",alignment="-1",col_name="current",colhdr="Current"}],
body=[variable={name="$trace_timestamp",initial="0"}
      variable={name="$foo",initial="10",current="15"}]}
(gdb)
```

-trace-save

Synopsis

```
-trace-save [ -r ] [ -ctf ] filename
```

Saves the collected trace data to *filename*. Without the ‘`-r`’ option, the data is downloaded from the target and saved in a local file. With the ‘`-r`’ option the target is asked to perform the save.

By default, this command will save the trace in the tfile format. You can supply the optional ‘`-ctf`’ argument to save it the CTF format. See Section 13.4 [Trace Files], page 209, for more information about CTF.

GDB Command

The corresponding GDB command is ‘`tsave`’.

-trace-start

Synopsis

```
-trace-start
```

Starts a tracing experiment. The result of this command does not have any fields.

GDB Command

The corresponding GDB command is ‘`tstart`’.

-trace-status

Synopsis

```
-trace-status
```

Example

```
(gdb)
-symbol-info-functions
^done,symbols=
  {debug=
    [{filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      symbols=[{line="36", name="f4", type="void (int *)",
                description="void f4(int *);"},
               {line="42", name="main", type="int ()",
                description="int main();"},
               {line="30", name="f1", type="my_int_t (int, int)",
                description="static my_int_t f1(int, int);"}]},
     {filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      symbols=[{line="33", name="f2", type="float (another_float_t)",
                description="float f2(another_float_t);"},
               {line="39", name="f3", type="int (another_int_t)",
                description="int f3(another_int_t);"},
               {line="27", name="f1", type="another_float_t (int)",
                description="static another_float_t f1(int);"}]}]}
(gdb)
-symbol-info-functions --name f1
^done,symbols=
  {debug=
    [{filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      symbols=[{line="30", name="f1", type="my_int_t (int, int)",
                description="static my_int_t f1(int, int);"}]},
     {filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      symbols=[{line="27", name="f1", type="another_float_t (int)",
                description="static another_float_t f1(int);"}]}]}
(gdb)
-symbol-info-functions --type void
^done,symbols=
  {debug=
    [{filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      symbols=[{line="36", name="f4", type="void (int *)",
                description="void f4(int *);"}]}]}
```

```
(gdb)
-symbol-info-functions --include-nondebug
^done,symbols=
  {debug=
    [{filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      symbols=[{line="36", name="f4", type="void (int *)",
                description="void f4(int *);"},
               {line="42", name="main", type="int ()",
                description="int main();"},
               {line="30", name="f1", type="my_int_t (int, int)",
                description="static my_int_t f1(int, int);"}]},
     {filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      symbols=[{line="33", name="f2", type="float (another_float_t)",
                description="float f2(another_float_t);"},
               {line="39", name="f3", type="int (another_int_t)",
                description="int f3(another_int_t);"},
               {line="27", name="f1", type="another_float_t (int)",
                description="static another_float_t f1(int);"}]}],
   nondebug=
    [{address="0x0000000000400398",name="_init"},
     {address="0x00000000004003b0",name="_start"},
      ...
    ]}
```

The `-symbol-info-module-functions` Command

Synopsis

```
-symbol-info-module-functions [--module module_regexp]
                              [--name name_regexp]
                              [--type type_regexp]
```

Return a list containing the names of all known functions within all know Fortran modules. The functions are grouped by source file and containing module, and shown with the line number on which each function is defined.

The option `--module` only returns results for modules matching *module_regexp*. The option `--name` only returns functions whose name matches *name_regexp*, and `--type` only returns functions whose type matches *type_regexp*.

GDB Command

The corresponding GDB command is '`info module functions`'.

Example

```
(gdb)
-symbol-info-module-functions
^done,symbols=
  [{module="mod1",
    files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
            fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
            symbols=[{line="21",name="mod1::check_all",type="void (void)",
                      description="void mod1::check_all(void);"}]}]},
    {module="mod2",
     files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
             fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
             symbols=[{line="30",name="mod2::check_var_i",type="void (void)",
                       description="void mod2::check_var_i(void);"}]}]},
    {module="mod3",
     files=[{filename="/projec/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
             fullname="/projec/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
             symbols=[{line="21",name="mod3::check_all",type="void (void)",
                       description="void mod3::check_all(void);"},
                      {line="27",name="mod3::check_mod2",type="void (void)",
                       description="void mod3::check_mod2(void);"}]}]},
    {module="modmany",
     files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
             fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
             symbols=[{line="35",name="modmany::check_some",type="void (void)",
                       description="void modmany::check_some(void);"}]}]},
    {module="moduse",
     files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
             fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
             symbols=[{line="44",name="moduse::check_all",type="void (void)",
                       description="void moduse::check_all(void);"},
                      {line="49",name="moduse::check_var_x",type="void (void)",
                       description="void moduse::check_var_x(void);"}]}]}]
```

The `-symbol-info-module-variables` Command

Synopsis

```
-symbol-info-module-variables [--module module_regexp]
                              [--name name_regexp]
                              [--type type_regexp]
```

Return a list containing the names of all known variables within all know Fortran modules. The variables are grouped by source file and containing module, and shown with the line number on which each variable is defined.

The option `--module` only returns results for modules matching *module_regexp*. The option `--name` only returns variables whose name matches *name_regexp*, and `--type` only returns variables whose type matches *type_regexp*.

GDB Command

The corresponding GDB command is '`info module variables`'.

Example

```
(gdb)
-symbol-info-module-variables
^done,symbols=
  [{module="mod1",
    files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
            fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
            symbols=[{line="18",name="mod1::var_const",type="integer(kind=4)",
                      description="integer(kind=4) mod1::var_const;"},
                     {line="17",name="mod1::var_i",type="integer(kind=4)",
                      description="integer(kind=4) mod1::var_i;"}]}]},
   {module="mod2",
    files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
            fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules-2.f90",
            symbols=[{line="28",name="mod2::var_i",type="integer(kind=4)",
                      description="integer(kind=4) mod2::var_i;"}]}]},
   {module="mod3",
    files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
            fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
            symbols=[{line="18",name="mod3::mod1",type="integer(kind=4)",
                      description="integer(kind=4) mod3::mod1;"},
                     {line="17",name="mod3::mod2",type="integer(kind=4)",
                      description="integer(kind=4) mod3::mod2;"},
                     {line="19",name="mod3::var_i",type="integer(kind=4)",
                      description="integer(kind=4) mod3::var_i;"}]}]},
   {module="modmany",
    files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
            fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
            symbols=[{line="33",name="modmany::var_a",type="integer(kind=4)",
                      description="integer(kind=4) modmany::var_a;"},
                     {line="33",name="modmany::var_b",type="integer(kind=4)",
                      description="integer(kind=4) modmany::var_b;"},
                     {line="33",name="modmany::var_c",type="integer(kind=4)",
                      description="integer(kind=4) modmany::var_c;"},
                     {line="33",name="modmany::var_i",type="integer(kind=4)",
                      description="integer(kind=4) modmany::var_i;"}]}]},
   {module="moduse",
    files=[{filename="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
            fullname="/project/gdb/testsuite/gdb.mi/mi-fortran-modules.f90",
            symbols=[{line="42",name="moduse::var_x",type="integer(kind=4)",
                      description="integer(kind=4) moduse::var_x;"},
                     {line="42",name="moduse::var_y",type="integer(kind=4)",
                      description="integer(kind=4) moduse::var_y;"}]}]}]
```

The `-symbol-info-modules` Command

Synopsis

```
-symbol-info-modules [--name name_regexp]
                     [--max-results limit]
```

Return a list containing the names of all known Fortran modules. The modules are grouped by source file, and shown with the line number on which each modules is defined.

The option `--name` allows the modules returned to be filtered based the name of the module.

Example

```
(gdb)
-symbol-info-types
^done,symbols=
  {debug=
     [{filename="gdb.mi/mi-sym-info-1.c",
       fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
       symbols=[{name="float"},
                {name="int"},
                {line="27",name="typedef int my_int_t;"}]},
      {filename="gdb.mi/mi-sym-info-2.c",
       fullname="/project/gdb.mi/mi-sym-info-2.c",
       symbols=[{line="24",name="typedef float another_float_t;"},
                {line="23",name="typedef int another_int_t;"},
                {name="float"},
                {name="int"}]}]}
(gdb)
-symbol-info-types --name _int_
^done,symbols=
  {debug=
     [{filename="gdb.mi/mi-sym-info-1.c",
       fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
       symbols=[{line="27",name="typedef int my_int_t;"}]},
      {filename="gdb.mi/mi-sym-info-2.c",
       fullname="/project/gdb.mi/mi-sym-info-2.c",
       symbols=[{line="23",name="typedef int another_int_t;"}]}]}
```

The `-symbol-info-variables` Command

Synopsis

```
-symbol-info-variables [--include-nondebug]
                      [--type type_regexp]
                      [--name name_regexp]
                      [--max-results limit]
```

Return a list containing the names and types for all global variables taken from the debug information. The variables are grouped by source file, and shown with the line number on which each variable is defined.

The `--include-nondebug` option causes the output to include data symbols from the symbol table.

The options `--type` and `--name` allow the symbols returned to be filtered based on either the name of the variable, or the type of the variable.

The option `--max-results` restricts the command to return no more than *limit* results. If exactly *limit* results are returned then there might be additional results available if a higher limit is used.

GDB Command

The corresponding GDB command is '`info variables`'.

```
(gdb)
-symbol-info-variables --include-nondebug
^done,symbols=
  {debug=
    [{filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-1.c",
      symbols=[{line="25",name="global_f1",type="float",
                description="static float global_f1;"},
               {line="24",name="global_i1",type="int",
                description="static int global_i1;"}]},
     {filename="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      fullname="/project/gdb/testsuite/gdb.mi/mi-sym-info-2.c",
      symbols=[{line="21",name="global_f2",type="int",
                description="int global_f2;"},
               {line="20",name="global_i2",type="int",
                description="int global_i2;"},
               {line="19",name="global_f1",type="float",
                description="static float global_f1;"},
               {line="18",name="global_i1",type="int",
                description="static int global_i1;"}]}],
   nondebug=
    [{address="0x00000000004005d0",name="_IO_stdin_used"},
     {address="0x00000000004005d8",name="__dso_handle"}
      ...
    ]}
```

The `-symbol-list-lines` Command

Synopsis

```
-symbol-list-lines filename
```

Print the list of lines that contain code and their associated program addresses for the given source filename. The entries are sorted in ascending PC order.

GDB Command

There is no corresponding GDB command.

Example

```
(gdb)
-symbol-list-lines basics.c
^done,lines=[{pc="0x08048554",line="7"},{pc="0x0804855a",line="8"}]
(gdb)
```

27.21 GDB/MI File Commands

This section describes the GDB/MI commands to specify executable file names and to read in and obtain symbol table information.

The `-file-exec-and-symbols` Command

Synopsis

```
-file-exec-and-symbols file
```

Specify the executable file to be debugged. This file is the one from which the symbol table is also read. If no file is specified, the command clears the executable and symbol

information. If breakpoints are set when using this command with no arguments, GDB will produce error messages. Otherwise, no output is produced, except a completion notification.

GDB Command

The corresponding GDB command is '`file`'.

Example

```
(gdb)
-file-exec-and-symbols /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

The `-file-exec-file` Command

Synopsis

```
-file-exec-file file
```

Specify the executable file to be debugged. Unlike '`-file-exec-and-symbols`', the symbol table is *not* read from this file. If used without argument, GDB clears the information about the executable file. No output is produced, except a completion notification.

GDB Command

The corresponding GDB command is '`exec-file`'.

Example

```
(gdb)
-file-exec-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

The `-file-list-exec-source-file` Command

Synopsis

```
-file-list-exec-source-file
```

List the line number, the current source file, and the absolute path to the current source file for the current executable. The macro information field has a value of '`1`' or '`0`' depending on whether or not the file includes preprocessor macro information.

GDB Command

The GDB equivalent is '`info source`'

Example

```
(gdb)
123-file-list-exec-source-file
123^done,line="1",file="foo.c",fullname="/home/bar/foo.c,macro-info="1"
(gdb)
```

The `-file-list-exec-source-files` Command

Synopsis

```
-file-list-exec-source-files [ --group-by-objfile ]
                             [ --dirname | --basename ]
                             [ -- ]
                             [ regexp ]
```

This command returns information about the source files GDB knows about, it will output both the filename and fullname (absolute file name) of a source file, though the fullname can be elided if this information is not known to GDB.

With no arguments this command returns a list of source files. Each source file is represented by a tuple with the fields; *file*, *fullname*, and *debug-fully-read*. The *file* is the display name for the file, while *fullname* is the absolute name of the file. The *fullname* field can be elided if the absolute name of the source file can't be computed. The field *debug-fully-read* will be a string, either `true` or `false`. When `true`, this indicates the full debug information for the compilation unit describing this file has been read in. When `false`, the full debug information has not yet been read in. While reading in the full debug information it is possible that GDB could become aware of additional source files.

The optional *regexp* can be used to filter the list of source files returned. The *regexp* will be matched against the full source file name. The matching is case-sensitive, except on operating systems that have case-insensitive filesystem (e.g., MS-Windows). '`--`' can be used before *regexp* to prevent GDB interpreting *regexp* as a command option (e.g. if *regexp* starts with '`-`').

If `--dirname` is provided, then *regexp* is matched only against the directory name of each source file. If `--basename` is provided, then *regexp* is matched against the basename of each source file. Only one of `--dirname` or `--basename` may be given, and if either is given then *regexp* is required.

If `--group-by-objfile` is used then the format of the results is changed. The results will now be a list of tuples, with each tuple representing an object file (executable or shared library) loaded into GDB. The fields of these tuples are; *filename*, *debug-info*, and *sources*. The *filename* is the absolute name of the object file, *debug-info* is a string with one of the following values:

`none` This object file has no debug information.

`partially-read`
: This object file has debug information, but it is not fully read in yet. When it is read in later, GDB might become aware of additional source files.

`fully-read`
: This object file has debug information, and this information is fully read into GDB. The list of source files is complete.

The *sources* is a list or tuples, with each tuple describing a single source file with the same fields as described previously. The *sources* list can be empty for object files that have no debug information.

GDB Command

The GDB equivalent is '`info sources`'. `gdbtk` has an analogous command '`gdb_listfiles`'.

```
{filename="/lib64/libc.so.6",
 debug-info="none",
 sources=[]}]
```

The -file-list-shared-libraries Command

Synopsis

```
-file-list-shared-libraries [ regexp ]
```

List the shared libraries in the program. With a regular expression *regexp*, only those libraries whose names match *regexp* are listed.

GDB Command

The corresponding GDB command is '`info shared`'. The fields have a similar meaning to the `=library-loaded` notification. The `ranges` field specifies the multiple segments belonging to this library. Each range has the following fields:

'`from`' The address defining the inclusive lower bound of the segment.

'`to`' The address defining the exclusive upper bound of the segment.

Example

```
(gdb)
-file-list-exec-source-files
^done,shared-libraries=[
{id="/lib/libfoo.so",target-name="/lib/libfoo.so",host-name="/lib/libfoo.so",symbols-loaded="1",thread-
group="i1",ranges=[{from="0x72815989",to="0x728162c0"}]},
{id="/lib/libbar.so",target-name="/lib/libbar.so",host-name="/lib/libbar.so",symbols-loaded="1",thread-
group="i1",ranges=[{from="0x76ee48c0",to="0x76ee9160"}]}]
(gdb)
```

The -file-symbol-file Command

Synopsis

```
-file-symbol-file file
```

Read symbol table info from the specified *file* argument. When used without arguments, clears GDB's symbol table info. No output is produced, except for a completion notification.

GDB Command

The corresponding GDB command is '`symbol-file`'.

Example

```
(gdb)
-file-symbol-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

27.22 GDB/MI Target Manipulation Commands

The -target-attach Command

Synopsis

```
-target-attach pid | gid | file
```

Attach to a process *pid* or a file *file* outside of GDB, or a thread group *gid*. If attaching to a thread group, the id previously returned by '`-list-thread-groups --available`' must be used.

GDB Command

The corresponding GDB command is '`attach`'.

Example

```
(gdb)
-target-attach 34
=thread-created,id="1"
*stopped,thread-id="1",frame={addr="0xb7f7e410",func="bar",args=[]}
^done
(gdb)
```

The `-target-detach` Command

Synopsis

```
-target-detach [ pid | gid ]
```

Detach from the remote target which normally resumes its execution. If either *pid* or *gid* is specified, detaches from either the specified process, or specified thread group. There's no output.

GDB Command

The corresponding GDB command is '`detach`'.

Example

```
(gdb)
-target-detach
^done
(gdb)
```

The `-target-disconnect` Command

Synopsis

```
-target-disconnect
```

Disconnect from the remote target. There's no output and the target is generally not resumed.

GDB Command

The corresponding GDB command is '`disconnect`'.

Example

```
(gdb)
-target-disconnect
^done
(gdb)
```

The `-target-download` Command

Synopsis

```
-target-download
```

Loads the executable onto the remote target. It prints out an update message every half second, which includes the fields:

‘`section`’ The name of the section.

‘`section-sent`’
The size of what has been sent so far for that section.

‘`section-size‘`

```
+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)
```

GDB Command

No equivalent.

Example

N.A.

The `-target-flash-erase` Command

Synopsis

```
-target-flash-erase
```

Erases all known flash memory regions on the target.

The corresponding GDB command is '`flash-erase`'.

The output is a list of flash regions that have been erased, with starting addresses and memory region sizes.

```
(gdb)
-target-flash-erase
^done,erased-regions={address="0x0",size="0x40000"}
(gdb)
```

The `-target-select` Command

Synopsis

```
-target-select type parameters ...
```

Connect GDB to the remote target. This command takes two args:

'*type*' The type of target, for instance '`remote`', etc.

'*parameters*'
Device names, host names and the like. See Section 19.2 [Commands for Managing Targets], page 295, for more details.

The output is a connection notification, followed by the address at which the target program is, in the following form:

```
^connected,addr="address",func="function name",
  args=[arg list]
```

GDB Command

The corresponding GDB command is '`target`'.

Example

```
(gdb)
-target-select remote /dev/ttya
^connected,addr="0xfe00a300",func="??",args=[]
(gdb)
```

27.23 GDB/MI File Transfer Commands

The `-target-file-put` Command

Synopsis

```
-target-file-put hostfile targetfile
```

Copy file *hostfile* from the host system (the machine running GDB) to *targetfile* on the target system.

GDB Command

The corresponding GDB command is '`remote put`'.

Example

```
(gdb)
-target-file-put localfile remotefile
^done
(gdb)
```

The `-target-file-get` Command

Synopsis

```
-target-file-get targetfile hostfile
```

Copy file *targetfile* from the target system to *hostfile* on the host system.

GDB Command

The corresponding GDB command is '`remote get`'.

Example

```
(gdb)
-target-file-get remotefile localfile
^done
(gdb)
```

The `-target-file-delete` Command

Synopsis

```
-target-file-delete targetfile
```

Delete *targetfile* from the target system.

GDB Command

The corresponding GDB command is '`remote delete`'.

Example

```
(gdb)
-target-file-delete remotefile
^done
(gdb)
```

27.24 Ada Exceptions GDB/MI Commands

The `-info-ada-exceptions` Command

Synopsis

```
-info-ada-exceptions [ regexp]
```

List all Ada exceptions defined within the program being debugged. With a regular expression *regexp*, only those exceptions whose names match *regexp* are listed.

GDB Command

The corresponding GDB command is '`info exceptions`'.

Result

The result is a table of Ada exceptions. The following columns are defined for each exception:

'`name`' The name of the exception.

'`address`' The address of the exception.

Example

```
-info-ada-exceptions aint
^done,ada-exceptions={nr_rows="2",nr_cols="2",
hdr=[{width="1",alignment="-1",col_name="name",colhdr="Name"},
{width="1",alignment="-1",col_name="address",colhdr="Address"}],
```

```
body=[{name="constraint_error",address="0x0000000000613da0"},
{name="const.aint_global_e",address="0x0000000000613b00"}]}
```

Catching Ada Exceptions

The commands describing how to ask GDB to stop when a program raises an exception are described at Section 27.11.2 [Ada Exception GDB/MI Catchpoint Commands], page 565.

27.25 GDB/MI Support Commands

Since new commands and features get regularly added to GDB/MI, some commands are available to help front-ends query the debugger about support for these capabilities. Similarly, it is also possible to query GDB about target support of certain features.

The `-info-gdb-mi-command` Command

Synopsis

```
-info-gdb-mi-command cmd_name
```

Query support for the GDB/MI command named *cmd_name*.

Note that the dash (`-`) starting all GDB/MI commands is technically not part of the command name (see Section 27.4.1 [GDB/MI Input Syntax], page 540), and thus should be omitted in *cmd_name*. However, for ease of use, this command also accepts the form with the leading dash.

GDB Command

There is no corresponding GDB command.

Result

The result is a tuple. There is currently only one field:

'`exists`' This field is equal to `"true"` if the GDB/MI command exists, `"false"` otherwise.

Example

Here is an example where the GDB/MI command does not exist:

```
-info-gdb-mi-command unsupported-command
^done,command={exists="false"}
```

And here is an example where the GDB/MI command is known to the debugger:

```
-info-gdb-mi-command symbol-list-lines
^done,command={exists="true"}
```

The `-list-features` Command

Returns a list of particular features of the MI protocol that this version of gdb implements. A feature can be a command, or a new field in an output of some command, or even an important bugfix. While a frontend can sometimes detect presence of a feature at runtime, it is easier to perform detection at debugger startup.

The command returns a list of strings, with each string naming an available feature. Each returned string is just a name, it does not have any internal structure. The list of possible feature names is given below.

Example output:

```
(gdb) -list-features
^done,result=["feature1","feature2"]
```

The current list of features is:

'`frozen-varobjs`'
: Indicates support for the `-var-set-frozen` command, as well as possible presence of the `frozen` field in the output of `-varobj-create`.

'`pending-breakpoints`'
: Indicates support for the `-f` option to the `-break-insert` command.

'`python`'
: Indicates Python scripting support, Python-based pretty-printing commands, and possible presence of the '`display_hint`' field in the output of `-var-list-children`

'`thread-info`'
: Indicates support for the `-thread-info` command.

'`data-read-memory-bytes`'
: Indicates support for the `-data-read-memory-bytes` and the `-data-write-memory-bytes` commands.

'`breakpoint-notifications`'
: Indicates that changes to breakpoints and breakpoints created via the CLI will be announced via async records.

'`ada-task-info`'
: Indicates support for the `-ada-task-info` command.

'`language-option`'
: Indicates that all GDB/MI commands accept the `--language` option (see Section 27.3.1 [Context management], page 538).

'`info-gdb-mi-command`'
: Indicates support for the `-info-gdb-mi-command` command.

'`undefined-command-error-code`'
: Indicates support for the "undefined-command" error code in error result records, produced when trying to execute an undefined GDB/MI command (see Section 27.7.1 [GDB/MI Result Records], page 544).

'`exec-run-start-option`'
: Indicates that the `-exec-run` command supports the `--start` option (see Section 27.15 [GDB/MI Program Execution], page 574).

'`data-disassemble-a-option`'
: Indicates that the `-data-disassemble` command supports the `-a` option (see Section 27.18 [GDB/MI Data Manipulation], page 597).

The `-list-target-features` Command

Returns a list of particular features that are supported by the target. Those features affect the permitted MI commands, but unlike the features reported by the `-list-features`

command, the features depend on which target GDB is using at the moment. Whenever a target can change, due to commands such as `-target-select`, `-target-attach` or `-exec-run`, the list of target features may change, and the frontend should obtain it again. Example output:

```
(gdb) -list-target-features
^done,result=["async"]
```

The current list of features is:

'`async`' Indicates that the target is capable of asynchronous command execution, which means that GDB will accept further commands while the target is running.

'`reverse`' Indicates that the target is capable of reverse execution. See Chapter 6 [Reverse Execution], page 97, for more information.

27.26 Miscellaneous GDB/MI Commands

The `-gdb-exit` Command

Synopsis

```
-gdb-exit
```

Exit GDB immediately.

GDB Command

Approximately corresponds to '`quit`'.

Example

```
(gdb)
-gdb-exit
^exit
```

The `-gdb-set` Command

Synopsis

```
-gdb-set
```

Set an internal GDB variable.

GDB Command

The corresponding GDB command is '`set`'.

Example

```
(gdb)
-gdb-set $foo=3
^done
(gdb)
```

The `-gdb-show` Command

Synopsis

```
-gdb-show
```

Show the current value of a GDB variable.

GDB Command

The corresponding GDB command is '`show`'.

Example

```
(gdb)
-gdb-show annotate
^done,value="0"
(gdb)
```

The `-gdb-version` Command

Synopsis

```
-gdb-version
```

Show version information for GDB. Used mostly in testing.

GDB Command

The GDB equivalent is '`show version`'. GDB by default shows this information when you start an interactive session.

Example

```
(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)
```

The `-list-thread-groups` Command

Synopsis

```
-list-thread-groups [ --available ] [ --recurse 1 ] [ group ... ]
```

Lists thread groups (see Section 27.3.3 [Thread groups], page 540). When a single thread group is passed as the argument, lists the children of that group. When several thread group are passed, lists information about those thread groups. Without any parameters, lists information about all top-level thread groups.

Normally, thread groups that are being debugged are reported. With the '`--available`' option, GDB reports thread groups available on the target.

The output of this command may have either a '`threads`' result or a '`groups`' result. The '`thread`' result has a list of tuples as value, with each tuple describing a thread (see Section 27.7.6 [GDB/MI Thread Information], page 552). The '`groups`' result has a list of tuples as value, each tuple describing a thread group. If top-level groups are requested

(that is, no parameter is passed), or when several groups are passed, the output always has a '`groups`' result. The format of the '`group`' result is described below.

To reduce the number of roundtrips it's possible to list thread groups together with their children, by passing the '`--recurse`' option and the recursion depth. Presently, only recursion depth of 1 is permitted. If this option is present, then every reported thread group will also include its children, either as '`group`' or '`threads`' field.

In general, any combination of option and parameters is permitted, with the following caveats:

- When a single thread group is passed, the output will typically be the '`threads`' result. Because threads may not contain anything, the '`recurse`' option will be ignored.
- When the '`--available`' option is passed, limited information may be available. In particular, the list of threads of a process might be inaccessible. Further, specifying specific thread groups might not give any performance advantage over listing all thread groups. The frontend should assume that '`-list-thread-groups --available`' is always an expensive operation and cache the results.

The '`groups`' result is a list of tuples, where each tuple may have the following fields:

`id` Identifier of the thread group. This field is always present. The identifier is an opaque string; frontends should not try to convert it to an integer, even though it might look like one.

`type` The type of the thread group. At present, only '`process`' is a valid type.

`pid` The target-specific process identifier. This field is only present for thread groups of type '`process`' and only if the process exists.

`exit-code`
The exit code of this group's last exited thread, formatted in octal. This field is only present for thread groups of type '`process`' and only if the process is not running.

`num_children`
The number of children this thread group has. This field may be absent for an available thread group.

`threads` This field has a list of tuples as value, each tuple describing a thread. It may be present if the '`--recurse`' option is specified, and it's actually possible to obtain the threads.

`cores` This field is a list of integers, each identifying a core that one thread of the group is running on. This field may be absent if such information is not available.

`executable`
The name of the executable file that corresponds to this thread group. The field is only present for thread groups of type '`process`', and only if there is a corresponding executable file.

Example

```
(gdb)
-list-thread-groups
```

```
      item={col0="sockets",col1="Listing of all internet-domain sockets",
            col2="Sockets"},
      item={col0="threads",col1="Listing of all threads",
            col2="Threads"}]
(gdb)
-info-os processes
^done,OSDataTable={nr_rows="190",nr_cols="4",
hdr=[{width="10",alignment="-1",col_name="col0",colhdr="pid"},
     {width="10",alignment="-1",col_name="col1",colhdr="user"},
     {width="10",alignment="-1",col_name="col2",colhdr="command"},
     {width="10",alignment="-1",col_name="col3",colhdr="cores"}],
body=[item={col0="1",col1="root",col2="/sbin/init",col3="0"},
      item={col0="2",col1="root",col2="[kthreadd]",col3="1"},
      item={col0="3",col1="root",col2="[ksoftirqd/0]",col3="0"},
      ...
      item={col0="26446",col1="stan",col2="bash",col3="0"},
      item={col0="28152",col1="stan",col2="bash",col3="1"}]}
(gdb)
```

(Note that the MI output here includes a `"Title"` column that does not appear in command-line `info os`; this column is useful for MI clients that want to enumerate the types of data, such as in a popup menu, but is needless clutter on the command line, and `info os` omits it.)

The `-add-inferior` Command

Synopsis

```
-add-inferior [ --no-connection ]
```

Creates a new inferior (see Section 4.9 [Inferiors Connections and Programs], page 40). The created inferior is not associated with any executable. Such association may be established with the '`-file-exec-and-symbols`' command (see Section 27.21 [GDB/MI File Commands], page 620).

By default, the new inferior begins connected to the same target connection as the current inferior. For example, if the current inferior was connected to `gdbserver` with `target remote`, then the new inferior will be connected to the same `gdbserver` instance. The '`--no-connection`' option starts the new inferior with no connection yet. You can then for example use the `-target-select remote` command to connect to some other `gdbserver` instance, use `-exec-run` to spawn a local program, etc.

The command response always has a field, *inferior*, whose value is the identifier of the thread group corresponding to the new inferior.

An additional section field, *connection*, is optional. This field will only exist if the new inferior has a target connection. If this field exists, then its value will be a tuple containing the following fields:

'`number`' The number of the connection used for the new inferior.

'`name`' The name of the connection type used for the new inferior.

GDB Command

The corresponding GDB command is '`add-inferior`' (see ['`add-inferior`'], page 42).

Example

```
(gdb)
-add-inferior
^done,inferior="i3"
```

The `-interpreter-exec` Command

Synopsis

```
-interpreter-exec interpreter command
```

Execute the specified *command* in the given *interpreter*.

GDB Command

The corresponding GDB command is '`interpreter-exec`'.

Example

```
(gdb)
-interpreter-exec console "break main"
&"During symbol reading, couldn't parse type; debugger out of date?.\n"
&"During symbol reading, bad structure-type format.\n"
~"Breakpoint 1 at 0x8074fc6: file ../../src/gdb/main.c, line 743.\n"
^done
(gdb)
```

The `-inferior-tty-set` Command

Synopsis

```
-inferior-tty-set /dev/pts/1
```

Set terminal for future runs of the program being debugged.

GDB Command

The corresponding GDB command is '`set inferior-tty`' /dev/pts/1.

Example

```
(gdb)
-inferior-tty-set /dev/pts/1
^done
(gdb)
```

The `-inferior-tty-show` Command

Synopsis

```
-inferior-tty-show
```

Show terminal for future runs of program being debugged.

GDB Command

The corresponding GDB command is '`show inferior-tty`'.

Example

```
(gdb)
-inferior-tty-set /dev/pts/1
^done
(gdb)
-inferior-tty-show
^done,inferior_tty_terminal="/dev/pts/1"
(gdb)
```

The `-enable-timings` Command

Synopsis

```
-enable-timings [yes | no]
```

Toggle the printing of the wallclock, user and system times for an MI command as a field in its output. This command is to help frontend developers optimize the performance of their code. No argument is equivalent to '`yes`'.

GDB Command

No equivalent.

Example

```
(gdb)
-enable-timings
^done
(gdb)
-break-insert main
^done,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x080484ed",func="main",file="myprog.c",
fullname="/home/nickrob/myprog.c",line="73",thread-groups=["i1"],
times="0"},
time={wallclock="0.05185",user="0.00800",system="0.00000"}
(gdb)
-enable-timings no
^done
(gdb)
-exec-run
^running
(gdb)
*stopped,reason="breakpoint-hit",disp="keep",bkptno="1",thread-id="0",
frame={addr="0x080484ed",func="main",args=[{name="argc",value="1"},
{name="argv",value="0xbfb60364"}],file="myprog.c",
fullname="/home/nickrob/myprog.c",line="73",arch="i386:x86_64"}
(gdb)
```

The `-complete` Command

Synopsis

```
-complete command
```

Show a list of completions for partially typed CLI *command*.

This command is intended for GDB/MI frontends that cannot use two separate CLI and MI channels — for example: because of lack of PTYs like on Windows or because GDB is used remotely via a SSH connection.

Result

The result consists of two or three fields:

‘`completion`’
: This field contains the completed *command*. If *command* has no known completions, this field is omitted.

‘`matches`’
: This field contains a (possibly empty) array of matches. It is always present.

‘`max_completions_reached`’
: This field contains `1` if number of known completions is above `max-completions` limit (see Section 3.3 [Completion], page 24), otherwise it contains `0`. It is always present.

GDB Command

The corresponding GDB command is ‘`complete`’.

Example

```
(gdb)
-complete br
^done,completion="break",
      matches=["break","break-range"],
      max_completions_reached="0"
(gdb)
-complete "b ma"
^done,completion="b ma",
      matches=["b madvise","b main"],max_completions_reached="0"
(gdb)
-complete "b push_b"
^done,completion="b push_back(",
      matches=[
       "b A::push_back(void*)",
       "b std::string::push_back(char)",
       "b std::vector<int, std::allocator<int> >::push_back(int&&)"],
      max_completions_reached="0"
(gdb)
-complete "nonexist"
^done,matches=[],max_completions_reached="0"
(gdb)
```

28 gdb Annotations

This chapter describes annotations in gdb. Annotations were designed to interface gdb to graphical user interfaces or other similar programs which want to interact with gdb at a relatively high level.

The annotation mechanism has largely been superseded by gdb/mi (see Chapter 27 [GDB/MI], page 537).

28.1 What is an Annotation?

Annotations start with a newline character, two `control-z' characters, and the name of the annotation. If there is no additional information associated with this annotation, the name of the annotation is followed immediately by a newline. If there is additional information, the name of the annotation is followed by a space, the additional information, and a newline. The additional information cannot contain newline characters.

Any output not beginning with a newline and two `control-z' characters denotes literal output from gdb. Currently there is no need for gdb to output a newline followed by two `control-z' characters, but if there was such a need, the annotations could be extended with an `escape' annotation which means those three characters as output.

The annotation level, which is specified using the --annotate command line option (see Section 2.1.2 [Mode Options], page 13), controls how much information gdb prints together with its prompt, values of expressions, source lines, and other types of output. Level 0 is for no annotations, level 1 is for use when gdb is run as a subprocess of GNU Emacs, level 3 is the maximum annotation suitable for programs that control gdb, and level 2 annotations have been made obsolete (see Section "Limitations of the Annotation Interface" in GDB's Obsolete Annotations).

set annotate level
: The gdb command set annotate sets the level of annotations to the specified level.

show annotate
: Show the current annotation level.

This chapter describes level 3 annotations.

A simple example of starting up gdb with annotations is:

```
$ gdb --annotate=3
GNU gdb 6.0
Copyright 2003 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 "i386-pc-linux-gnu"

^Z^Zpre-prompt
(gdb)
^Z^Zprompt
quit
```

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

  However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

  A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GDB it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

  And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

Appendix A In Memoriam

The GDB project mourns the loss of the following long-time contributors:

`Fred Fish` Fred was a long-standing contributor to GDB (1991-2006), and to Free Software in general. Outside of GDB, he was known in the Amiga world for his series of Fish Disks, and the GeekGadget project.

`Michael Snyder`
Michael was one of the Global Maintainers of the GDB project, with contributions recorded as early as 1996, until 2011. In addition to his day to day participation, he was a large driving force behind adding Reverse Debugging to GDB.

Beyond their technical contributions to the project, they were also enjoyable members of the Free Software Community. We will miss them.

Appendix B Formatting Documentation

The GDB 4 release includes an already-formatted reference card, ready for printing with PostScript or Ghostscript, in the `gdb` subdirectory of the main source directory[1]. If you can use PostScript or Ghostscript with your printer, you can print the reference card immediately with `refcard.ps`.

The release also includes the source for the reference card. You can format it, using TEX, by typing:

```
make refcard.dvi
```

The GDB reference card is designed to print in *landscape* mode on US "letter" size paper; that is, on a sheet 11 inches wide by 8.5 inches high. You will need to specify this form of printing as an option to your DVI output program.

All the documentation for GDB comes as part of the machine-readable distribution. The documentation is written in Texinfo format, which is a documentation system that uses a single source file to produce both on-line information and a printed manual. You can use one of the Info formatting commands to create the on-line version of the documentation and TEX (or `texi2roff`) to typeset the printed version.

GDB includes an already formatted copy of the on-line Info version of this manual in the `gdb` subdirectory. The main Info file is `gdb-AlmaLinux 12.1-5.el9/gdb/gdb.info`, and it refers to subordinate files matching '`gdb.info*`' in the same directory. If necessary, you can print out these files, or read them with any editor; but they are easier to read using the `info` subsystem in GNU Emacs or the standalone `info` program, available as part of the GNU Texinfo distribution.

If you want to format these Info files yourself, you need one of the Info formatting programs, such as `texinfo-format-buffer` or `makeinfo`.

If you have `makeinfo` installed, and are in the top level GDB source directory (`gdb-AlmaLinux 12.1-5.el9`, in the case of version AlmaLinux 12.1-5.el9), you can make the Info file by typing:

```
cd gdb
make gdb.info
```

If you want to typeset and print copies of this manual, you need TEX, a program to print its DVI output files, and `texinfo.tex`, the Texinfo definitions file.

TEX is a typesetting program; it does not print files directly, but produces output files called DVI files. To print a typeset document, you need a program to print DVI files. If your system has TEX installed, chances are it has such a program. The precise command to use depends on your system; *`lpr -d`* is common; another (for PostScript devices) is *`dvips`*. The DVI print command may require a file name without any extension or a '`.dvi`' extension.

TEX also requires a macro definitions file called `texinfo.tex`. This file tells TEX how to typeset a document written in Texinfo format. On its own, TEX cannot either read or typeset a Texinfo file. `texinfo.tex` is distributed with GDB and is located in the `gdb-`*`version-number`*`/texinfo` directory.

[1] In `gdb-AlmaLinux 12.1-5.el9/gdb/refcard.ps` of the version AlmaLinux 12.1-5.el9 release.

You should run the `configure` script from the top directory in the source tree, the `gdb-`*`version-number`* directory. If you run `configure` from one of the subdirectories, you will configure only that subdirectory. That is usually not what you want. In particular, if you run the first `configure` from the `gdb` subdirectory of the `gdb-`*`version-number`* directory, you will omit the configuration of `bfd`, `readline`, and other sibling directories of the `gdb` subdirectory. This leads to build errors about missing include files such as `bfd/bfd.h`.

You can install GDB anywhere. The best way to do this is to pass the `--prefix` option to `configure`, and then install it with `make install`.

C.3 Compiling GDB in Another Directory

If you want to run GDB versions for several host or target machines, you need a different `gdb` compiled for each combination of host and target. `configure` is designed to make this easy by allowing you to generate each configuration in a separate subdirectory, rather than in the source directory. If your `make` program handles the '`VPATH`' feature (GNU `make` does), running `make` in each of these directories builds the `gdb` program specified there.

To build `gdb` in a separate directory, run `configure` with the '`--srcdir`' option to specify where to find the source. (You also need to specify a path to find `configure` itself from your working directory. If the path to `configure` would be the same as the argument to '`--srcdir`', you can leave out the '`--srcdir`' option; it is assumed.)

For example, with version AlmaLinux 12.1-5.el9, you can build GDB in a separate directory for a Sun 4 like this:

```
cd gdb-AlmaLinux 12.1-5.el9
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-AlmaLinux 12.1-5.el9/configure
make
```

When `configure` builds a configuration using a remote source directory, it creates a tree for the binaries with the same structure (and using the same names) as the tree under the source directory. In the example, you'd find the Sun 4 library `libiberty.a` in the directory `gdb-sun4/libiberty`, and GDB itself in `gdb-sun4/gdb`.

Make sure that your path to the `configure` script has just one instance of `gdb` in it. If your path to `configure` looks like `../gdb-AlmaLinux 12.1-5.el9/gdb/configure`, you are configuring only one subdirectory of GDB, not the whole package. This leads to build errors about missing include files such as `bfd/bfd.h`.

One popular reason to build several GDB configurations in separate directories is to configure GDB for cross-compiling (where GDB runs on one machine—the *host*—while debugging programs that run on another machine—the *target*). You specify a cross-debugging target by giving the '`--target=`*`target`*' option to `configure`.

When you run `make` to build a program or library, you must run it in a configured directory—whatever directory you were in when you called `configure` (or one of its subdirectories).

The `Makefile` that `configure` generates in each source directory also runs recursively. If you type `make` in a source directory such as `gdb-AlmaLinux 12.1-5.el9` (or in a separate configured directory configured with '`--srcdir=`*`dirname`*`/gdb-AlmaLinux 12.1-5.el9`'), you will build all the required libraries, and then build GDB.

When you have multiple hosts or targets configured in separate directories, you can run `make` on them in parallel (for example, if they are NFS-mounted on each of the hosts); they

`--with-python`[`=`*python*]
: Build GDB with Python scripting support. (Done by default if libpython is present and found at configure time.) Python makes GDB scripting much more powerful than the restricted CLI scripting language. If your host does not have Python installed, you can find it on 'http://www.python.org/download/'. The oldest version of Python supported by GDB is 2.6. The optional argument *python* is used to find the Python headers and libraries. It can be either the name of a Python executable, or the name of the directory in which Python is installed.

`--with-guile[=GUILE]'`
: Build GDB with GNU Guile scripting support. (Done by default if libguile is present and found at configure time.) If your host does not have Guile installed, you can find it at 'https://www.gnu.org/software/guile/'. The optional argument GUILE can be a version number, which will cause `configure` to try to use that version of Guile; or the file name of a `pkg-config` executable, which will be queried to find the information needed to compile and link against Guile.

`--without-included-regex`
: Don't use the regex library included with GDB (as part of the libiberty library). This is the default on hosts with version 2 of the GNU C library.

`--with-sysroot=`*dir*
: Use *dir* as the default system root directory for libraries whose file names begin with `/lib`' or `/usr/lib'`. (The value of *dir* can be modified at run time by using the `set sysroot` command.) If *dir* is under the GDB configured prefix (set with `--prefix` or `--exec-prefix` `options`, the default system root will be automatically adjusted if and when GDB is moved to a different location.

`--with-system-gdbinit=`*file*
: Configure GDB to automatically load a system-wide init file. *file* should be an absolute file name. If *file* is in a directory under the configured prefix, and GDB is moved to another location after being built, the location of the system-wide init file will be adjusted accordingly.

`--with-system-gdbinit-dir=`*directory*
: Configure GDB to automatically load init files from a system-wide directory. *directory* should be an absolute directory name. If *directory* is in a directory under the configured prefix, and GDB is moved to another location after being built, the location of the system-wide init directory will be adjusted accordingly.

`--enable-build-warnings`
: When building the GDB sources, ask the compiler to warn about any code which looks even vaguely suspicious. It passes many different warning flags, depending on the exact version of the compiler you are using.

`--enable-werror`
: Treat compiler warnings as errors. It adds the `-Werror` flag to the compiler, which will fail the compilation if the compiler outputs any warning messages.

`--enable-ubsan`
: Enable the GCC undefined behavior sanitizer. This is disabled by default, but passing `--enable-ubsan=yes` or `--enable-ubsan=auto` to `configure` will enable it. The undefined behavior sanitizer checks for C++ undefined behavior. It has a performance cost, so if you are looking at GDB's performance, you should disable it. The undefined behavior sanitizer was first introduced in GCC 4.9.

C.6 System-wide configuration and settings

GDB can be configured to have a system-wide init file and a system-wide init file directory; this file and files in that directory (if they have a recognized file extension) will be read and executed at startup (see Section 2.1.3 [What GDB does during startup], page 16).

Here are the corresponding configure options:

`--with-system-gdbinit=`*file*
: Specify that the default location of the system-wide init file is *file*.

`--with-system-gdbinit-dir=`*directory*
: Specify that the default location of the system-wide init file directory is *directory*.

If GDB has been configured with the option `--prefix=$prefix`, they may be subject to relocation. Two possible cases:

- If the default location of this init file/directory contains `$prefix`, it will be subject to relocation. Suppose that the configure options are `--prefix=$prefix --with-system-gdbinit=$prefix/etc/gdbinit`; if GDB is moved from `$prefix` to `$install`, the system init file is looked for as `$install/etc/gdbinit` instead of `$prefix/etc/gdbinit`.
- By contrast, if the default location does not contain the prefix, it will not be relocated. E.g. if GDB has been configured with `--prefix=/usr/local --with-system-gdbinit=/usr/share/gdb/gdbinit`, then GDB will always look for `/usr/share/gdb/gdbinit`, wherever GDB is installed.

If the configured location of the system-wide init file (as given by the `--with-system-gdbinit` option at configure time) is in the data-directory (as specified by `--with-gdb-datadir` at configure time) or in one of its subdirectories, then GDB will look for the system-wide init file in the directory specified by the `--data-directory` command-line option. Note that the system-wide init file is only read once, during GDB initialization. If the data-directory is changed after GDB has started with the `set data-directory` command, the file will not be reread.

This applies similarly to the system-wide directory specified in `--with-system-gdbinit-dir`.

Any supported scripting language can be used for these init files, as long as the file extension matches the scripting language. To be interpreted as regular GDB commands, the files needs to have a `.gdb` extension.

C.6.1 Installed System-wide Configuration Scripts

The `system-gdbinit` directory, located inside the data-directory (as specified by `--with-gdb-datadir` at configure time) contains a number of scripts which can be used as system-wide init files. To automatically source those scripts at startup, GDB should be configured

`maint btrace packet-history`
: Print the raw branch trace packets that are used to compute the execution history for the '`record btrace`' command. Both the information and the format in which it is printed depend on the btrace recording format.

  `bts`
  : For the BTS recording format, print a list of blocks of sequential code. For each block, the following information is printed:

    Block number
    : Newer blocks have higher numbers. The oldest block has number zero.

    Lowest '`PC`'
    Highest '`PC`'

  `pt`
  : For the Intel Processor Trace recording format, print a list of Intel Processor Trace packets. For each packet, the following information is printed:

    Packet number
    : Newer packets have higher numbers. The oldest packet has number zero.

    Trace offset
    : The packet's offset in the trace stream.

    Packet opcode and payload

`maint btrace clear-packet-history`
: Discards the cached packet history printed by the '`maint btrace packet-history`' command. The history will be computed again when needed.

`maint btrace clear`
: Discard the branch trace data. The data will be fetched anew and the branch trace will be recomputed when needed.

  This implicitly truncates the branch trace to a single branch trace buffer. When updating branch trace incrementally, the branch trace available to GDB may be bigger than a single branch trace buffer.

`maint set btrace pt skip-pad`
`maint show btrace pt skip-pad`
: Control whether GDB will skip PAD packets when computing the packet history.

`maint info jit`
: Print information about JIT code objects loaded in the current inferior.

`set displaced-stepping`
`show displaced-stepping`
: Control whether or not GDB will do *displaced stepping* if the target supports it. Displaced stepping is a way to single-step over breakpoints without removing them from the inferior, by executing an out-of-line copy of the instruction that was originally at the breakpoint location. It is also known as out-of-line single-stepping.

`set displaced-stepping on`
: If the target architecture supports it, GDB will use displaced stepping to step over breakpoints.

`set displaced-stepping off`
: GDB will not use displaced stepping to step over breakpoints, even if such is supported by the target architecture.

`set displaced-stepping auto`
: This is the default mode. GDB will use displaced stepping only if non-stop mode is active (see Section 5.5.2 [Non-Stop Mode], page 91) and the target architecture supports displaced stepping.

`maint check-psymtabs`
: Check the consistency of currently expanded psymtabs versus symtabs. Use this to check, for example, whether a symbol is in one but not the other.

`maint check-symtabs`
: Check the consistency of currently expanded symtabs.

`maint expand-symtabs [`*regexp*`]`
: Expand symbol tables. If *regexp* is specified, only expand symbol tables for file names matching *regexp*.

`maint set catch-demangler-crashes [on|off]`
`maint show catch-demangler-crashes`
: Control whether GDB should attempt to catch crashes in the symbol name demangler. The default is to attempt to catch crashes. If enabled, the first time a crash is caught, a core file is created, the offending symbol is displayed and the user is presented with the option to terminate the current session.

`maint cplus first_component` *name*
: Print the first C++ class/namespace component of *name*.

`maint cplus namespace`
: Print the list of possible C++ namespaces.

`maint deprecate` *command* [*replacement*]
`maint undeprecate` *command*
: Deprecate or undeprecate the named *command*. Deprecated commands cause GDB to issue a warning when you use them. The optional argument *replacement* says which newer command should be used in favor of the deprecated one; if it is given, GDB will mention the replacement as part of the warning.

`maint dump-me`
: Cause a fatal signal in the debugger and force it to dump its core. This is supported only on systems which support aborting a program with the `SIGQUIT` signal.

`maint internal-error` [*message-text*]
`maint internal-warning` [*message-text*]
`maint demangler-warning` [*message-text*]
: Cause GDB to call the internal function `internal_error`, `internal_warning` or `demangler_warning` and hence behave as though an internal problem has

been detected. In addition to reporting the internal problem, these functions give the user the opportunity to either quit GDB or (for `internal_error` and `internal_warning`) create a core file of the current GDB session.

These commands take an optional parameter *message-text* that is used as the text of the error or warning message.

Here's an example of using `internal-error`:

```
(gdb) maint internal-error testing, 1, 2
.../maint.c:121: internal-error: testing, 1, 2
A problem internal to GDB has been detected.  Further
debugging may prove unreliable.
Quit this debugging session? (y or n) n
Create a core file? (y or n) n
(gdb)
```

`maint set internal-error` *action* [`ask|yes|no`]
`maint show internal-error` *action*
`maint set internal-warning` *action* [`ask|yes|no`]
`maint show internal-warning` *action*
`maint set demangler-warning` *action* [`ask|yes|no`]
`maint show demangler-warning` *action*

When GDB reports an internal problem (error or warning) it gives the user the opportunity to both quit GDB and create a core file of the current GDB session. These commands let you override the default behaviour for each particular *action*, described in the table below.

'`quit`' You can specify that GDB should always (yes) or never (no) quit. The default is to ask the user what to do.

'`corefile`'
You can specify that GDB should always (yes) or never (no) create a core file. The default is to ask the user what to do. Note that there is no `corefile` option for `demangler-warning`: demangler warnings always create a core file and this cannot be disabled.

`maint set internal-error backtrace` [`on|off`]
`maint show internal-error backtrace`
`maint set internal-warning backtrace` [`on|off`]
`maint show internal-warning backtrace`

When GDB reports an internal problem (error or warning) it is possible to have a backtrace of GDB printed to the standard error stream. This is '`on`' by default for `internal-error` and '`off`' by default for `internal-warning`.

`maint packet` *text*

If GDB is talking to an inferior via the serial protocol, then this command sends the string *text* to the inferior, and displays the response packet. GDB supplies the initial '`$`' character, the terminating '`#`' character, and the checksum.

Any non-printable characters in the reply are printed as escaped hex, e.g. '`\x00`', '`\x01`', etc.

`maint print architecture` [*file*]

Print the entire architecture configuration. The optional argument *file* names the file where the output goes.

`maint print c-tdesc` [`-single-feature`] [*file*]
: Print the target description (see Appendix G [Target Descriptions], page 769) as a C source file. By default, the target description is for the current target, but if the optional argument *file* is provided, that file is used to produce the description. The *file* should be an XML document, of the form described in Section G.2 [Target Description Format], page 769. The created source file is built into GDB when GDB is built again. This command is used by developers after they add or modify XML target descriptions.

  When the optional flag '`-single-feature`' is provided then the target description being processed (either the default, or from *file*) must only contain a single feature. The source file produced is different in this case.

`maint print xml-tdesc` [*file*]
: Print the target description (see Appendix G [Target Descriptions], page 769) as an XML file. By default print the target description for the current target, but if the optional argument *file* is provided, then that file is read in by GDB and then used to produce the description. The *file* should be an XML document, of the form described in Section G.2 [Target Description Format], page 769.

`maint check xml-descriptions` *dir*
: Check that the target descriptions dynamically created by GDB equal the descriptions created from XML files found in *dir*.

`maint check libthread-db`
: Run integrity checks on the current inferior's thread debugging library. This exercises all `libthread_db` functionality used by GDB on GNU/Linux systems, and by extension also exercises the `proc_service` functions provided by GDB that `libthread_db` uses. Note that parts of the test may be skipped on some platforms when debugging core files.

`maint print core-file-backed-mappings`
: Print the file-backed mappings which were loaded from a core file note. This output represents state internal to GDB and should be similar to the mappings displayed by the `info proc mappings` command.

`maint print dummy-frames`
: Prints the contents of GDB's internal dummy-frame stack.

```
(gdb) b add
...
(gdb) print add(2,3)
Breakpoint 2, add (a=2, b=3) at ...
58   return (a + b);
The program being debugged stopped while in a function called from GDB.
...
(gdb) maint print dummy-frames
0xa8206d8: id={stack=0xbfffe734,code=0xbfffe73f,!special}, ptid=process 9353
(gdb)
```

  Takes an optional file parameter.

`maint print registers` [*file*]
`maint print raw-registers` [*file*]
`maint print cooked-registers` [*file*]
`maint print register-groups` [*file*]
`maint print remote-registers` [*file*]

Print GDB's internal register data structures.

The command `maint print raw-registers` includes the contents of the raw register cache; the command `maint print cooked-registers` includes the (cooked) value of all registers, including registers which aren't available on the target nor visible to user; the command `maint print register-groups` includes the groups that each register is a member of; and the command `maint print remote-registers` includes the remote target's register numbers and offsets in the 'G' packets.

These commands take an optional parameter, a file name to which to write the information.

`maint print reggroups` [*file*]

Print GDB's internal register group data structures. The optional argument *file* tells to what file to write the information.

The register groups info looks like this:

```
(gdb) maint print reggroups
 Group      Type
 general    user
 float      user
 all        user
 vector     user
 system     user
 save       internal
 restore    internal
```

`maint flush register-cache`
`flushregs`

Flush the contents of the register cache and as a consequence the frame cache. This command is useful when debugging issues related to register fetching, or frame unwinding. The command `flushregs` is deprecated in favor of `maint flush register-cache`.

`maint flush source-cache`

Flush GDB's cache of source code file contents. After GDB reads a source file, and optionally applies styling (see Section 22.5 [Output Styling], page 349), the file contents are cached. This command clears that cache. The next time GDB wants to show lines from a source file, the content will be re-read.

This command is useful when debugging issues related to source code styling. After flushing the cache any source code displayed by GDB will be re-read and re-styled.

`maint print objfiles` [*regexp*]

Print a dump of all known object files. If *regexp* is specified, only print object files whose names match *regexp*. For each object file, this command prints its name, address in memory, and all of its psymtabs and symtabs.

`maint info selftests`
: List the selftests compiled in to GDB.

`maint set dwarf always-disassemble`
`maint show dwarf always-disassemble`
: Control the behavior of `info address` when using DWARF debugging information.tion.tion.

`maint set worker-threads`
`maint show worker-threads`
: Control the number of worker threads that may be used by GDB. On capable hosts, GDB may use multiple threads to speed up certain CPU-intensive operations, such as demangling symbol names. While the number of threads used by GDB may vary, this command can be used to set an upper bound on this number. The default is `unlimited`, which lets GDB choose a reasonable number. Note that this only controls worker threads started by GDB itself; libraries used by GDB may start threads of their own.

`maint set profile`
`maint show profile`
: Control profiling of GDB.

  Profiling will be disabled until you use the '`maint set profile`' command to enable it. When you enable profiling, the system will begin collecting timing and execution count data; when you disable profiling or exit GDB, the results will be written to a log file. Remember that if you use profiling, GDB will overwrite the profiling log file (often called `gmon.out`). If you have a record of important profiling data in a `gmon.out` file, be sure to move it to a safe location.

  Configuring with '`--enable-profiling`' arranges for GDB to be compiled with the '`-pg`' compiler option.

`maint set show-debug-regs`
`maint show show-debug-regs`
: Control whether to show variables that mirror the hardware debug registers. Use `on` to enable, `off` to disable. If enabled, the debug registers values are shown when GDB inserts or removes a hardware breakpoint or watchpoint, and when the inferior triggers a hardware-assisted breakpoint or watchpoint.

`maint set show-all-tib`
`maint show show-all-tib`
: Control whether to show all non zero areas within a 1k block starting at thread local base, when using the '`info w32 thread-information-block`' command.

`maint set target-async`
`maint show target-async`
: This controls whether GDB targets operate in synchronous or asynchronous mode (see Section 5.5.3 [Background Execution], page 92). Normally the default is asynchronous, if it is available; but this can be changed to more easily debug problems occurring only in synchronous mode.

`maint set target-non-stop`
`maint show target-non-stop`
: This controls whether GDB targets always operate in non-stop mode even if `set non-stop` is `off` (see Section 5.5.2 [Non-Stop Mode], page 91). The default is `auto`, meaning non-stop mode is enabled if supported by the target.

  `maint set target-non-stop auto`
  : This is the default mode. GDB controls the target in non-stop mode if the target supports it.

b. number of primary symbol tables
c. number of blocks in the blockvector

`maint set check-libthread-db [on|off]`
`maint show check-libthread-db`
: Control whether GDB should run integrity checks on inferior specific thread debugging libraries as they are loaded. The default is not to perform such checks. If any check fails GDB will unload the library and continue searching for a suitable candidate as described in [set libthread-db-search-path], page 48. For more information about the tests, see [maint check libthread-db], page 675.

`maint set gnu-source-highlight enabled` [`on|off`]
`maint show gnu-source-highlight enabled`
: Control whether GDB should use the GNU Source Highlight library for applying styling to source code (see Section 22.5 [Output Styling], page 349). This will be '`on`' by default if the GNU Source Highlight library is available. If the GNU Source Highlight library is not available, then this will be '`off`' by default, and attempting to change this value to '`on`' will give an error.

  If the GNU Source Highlight library is not being used, then GDB will use the Python Pygments package for source code styling, if it is available.

  This option is useful for debugging GDB's use of the Pygments library when GDB is linked against the GNU Source Highlight library.

`maint space` *value*
: An alias for `maint set per-command space`. A non-zero value enables it, zero disables it.

`maint time` *value*
: An alias for `maint set per-command time`. A non-zero value enables it, zero disables it.

`maint translate-address` [*section*] *addr*
: Find the symbol stored at the location specified by the address *addr* and an optional section name *section*. If found, GDB prints the name of the closest symbol and an offset from the symbol's location to the specified address. This is similar to the `info address` command (see Chapter 16 [Symbols], page 251), except that this command also allows to find symbols in other sections.

  If section was not specified, the section in which the symbol was found is also printed. For dynamically linked executables, the name of executable or shared library containing the symbol is printed as well.

`maint test-options require-delimiter`
`maint test-options unknown-is-error`
`maint test-options unknown-is-operand`
: These commands are used by the testsuite to validate the command options framework. The `require-delimiter` variant requires a double-dash delimiter to indicate end of options. The `unknown-is-error` and `unknown-is-operand` do not. The `unknown-is-error` variant throws an error on unknown option, while `unknown-is-operand` treats unknown options as the start of the command's operands. When run, the commands output the result of the processed

options. When completed, the commands store the internal result of completion in a variable exposed by the `maint show test-options-completion-result` command.

`maint show test-options-completion-result`
: Shows the result of completing the `maint test-options` subcommands. This is used by the testsuite to validate completion support in the command options framework.

`maint set test-settings` *kind*
`maint show test-settings` *kind*
: These are representative commands for each *kind* of setting type GDB supports. They are used by the testsuite for exercising the settings infrastructure.

`maint set backtrace-on-fatal-signal [on|off]`
`maint show backtrace-on-fatal-signal`
: When this setting is `on`, if GDB itself terminates with a fatal signal (e.g. SIGSEGV), then a limited backtrace will be printed to the standard error stream. This backtrace can be used to help diagnose crashes within GDB in situations where a user is unable to share a corefile with the GDB developers.

  If the functionality to provide this backtrace is not available for the platform on which GDB is running then this feature will be `off` by default, and attempting to turn this feature on will give an error.

  For platforms that do support creating the backtrace this feature is `on` by default.

`maint with` *setting* [*value*] [`--` *command*]
: Like the `with` command, but works with `maintenance set` variables. This is used by the testsuite to exercise the `with` command's infrastructure.

The following command is useful for non-interactive invocations of GDB, such as in the test suite.

`set watchdog` *nsec*
: Set the maximum number of seconds GDB will wait for the target operation to finish. If this time expires, GDB reports and error and the command is aborted.

`show watchdog`
: Show the current setting of the target wait timeout.

The binary data representation uses `7d` (ASCII '`}`') as an escape character. Any escaped byte is transmitted as the escape character followed by the original character XORed with `0x20`. For example, the byte `0x7d` would be transmitted as the two bytes `0x7d 0x5d`. The bytes `0x23` (ASCII '`#`'), `0x24` (ASCII '`$`'), and `0x7d` (ASCII '`}`') must always be escaped. Responses sent by the stub must also escape `0x2a` (ASCII '`*`'), so that it is not interpreted as the start of a run-length encoded sequence (described next).

Response *data* can be run-length encoded to save space. Run-length encoding replaces runs of identical characters with one instance of the repeated character, followed by a '`*`' and a repeat count. The repeat count is itself sent encoded, to avoid binary characters in *data*: a value of *n* is sent as `n+29`. For a repeat count greater or equal to 3, this produces a printable ASCII character, e.g. a space (ASCII code 32) for a repeat count of 3. (This is because run-length encoding starts to win for counts 3 or more.) Thus, for example, '`0* `' is a run-length encoding of "0000": the space character after '`*`' means repeat the leading `0` `32 - 29 = 3` more times.

The printable characters '`#`' and '`$`' or with a numeric value greater than 126 must not be used. Runs of six repeats ('`#`') or seven repeats ('`$`') can be expanded using a repeat count of only five ('`"`'). For example, '`00000000`' can be encoded as '`0*"00`'.

The error response returned for some packets includes a two character error number. That number is not well defined.

For any *command* not supported by the stub, an empty response ('`$#00`') should be returned. That way it is possible to extend the protocol. A newer GDB can tell if a packet is supported based on that response.

At a minimum, a stub is required to support the '`?`' command to tell GDB the reason for halting, '`g`' and '`G`' commands for register access, and the '`m`' and '`M`' commands for memory access. Stubs that only control single-threaded targets can implement run control with the '`c`' (continue) command, and if the target architecture supports hardware-assisted single-stepping, the '`s`' (step) command. Stubs that support multi-threading targets should support the '`vCont`' command. All other commands are optional.

E.2 Packets

The following table provides a complete list of all currently defined *command*s and their corresponding response *data*. See Section E.13 [File-I/O Remote Protocol Extension], page 738, for details about the File I/O extension of the remote protocol.

Each packet's description has a template showing the packet's overall syntax, followed by an explanation of the packet's meaning. We include spaces in some of the templates for clarity; these are not part of the packet's syntax. No GDB packet uses spaces to separate its components. For example, a template like '`foo` *`bar`* *`baz`*' describes a packet beginning with the three ASCII bytes '`foo`', followed by a *bar*, followed directly by a *baz*. GDB does not transmit a space character between the '`foo`' and the *bar*, or between the *bar* and the *baz*.

Several packets and replies include a *thread-id* field to identify a thread. Normally these are positive numbers with a target-specific interpretation, formatted as big-endian hex strings. A *thread-id* can also be a literal '`-1`' to indicate all threads, or '`0`' to pick any thread.

In addition, the remote protocol supports a multiprocess feature in which the *thread-id* syntax is extended to optionally include both process and thread ID fields, as '`p`*`pid`*`.`*`tid`*'.

The *pid* (process) and *tid* (thread) components each have the format described above: a positive number with target-specific interpretation formatted as a big-endian hex string, literal '`-1`' to indicate all processes or threads (respectively), or '`0`' to indicate an arbitrary process or thread. Specifying just a process, as '`p`*pid*', is equivalent to '`p`*pid*`.-1`'. It is an error to specify all processes but a specific thread, such as '`p-1.`*tid*'. Note that the '`p`' prefix is *not* used for those packets and replies explicitly documented to include a process ID, rather than a *thread-id*.

The multiprocess *thread-id* syntax extensions are only used if both GDB and the stub report support for the '`multiprocess`' feature using '`qSupported`'. See [multiprocess extensions], page 715, for more information.

Note that all packet forms beginning with an upper- or lower-case letter, other than those described here, are reserved for future use.

Here are the packet descriptions.

'`!`' Enable extended mode. In extended mode, the remote server is made persistent. The '`R`' packet is used to restart the program being debugged.

Reply:

'`OK`' The remote target both supports and has enabled extended mode.

'`?`' This is sent when connection is first established to query the reason the target halted. The reply is the same as for step and continue. This packet has a special interpretation when the target is in non-stop mode; see Section E.10 [Remote Non-Stop], page 737.

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

'`A` *arglen*`,`*argnum*`,`*arg*`,...`'

Initialized `argv[]` array passed into program. *arglen* specifies the number of bytes in the hex encoded byte stream *arg*. See `gdbserver` for more details.

Reply:

'`OK`' The arguments were set.

'`E` *NN*' An error occurred.

'`b` *baud*' (Don't use this packet; its behavior is not well-defined.) Change the serial line speed to *baud*.

JTC: *When does the transport layer state change? When it's received, or after the ACK is transmitted. In either case, there are problems if the command or the acknowledgment packet is dropped.*

Stan: *If people really wanted to add something like this, and get it working for the first time, they ought to modify ser-unix.c to send some kind of out-of-band message to a specially-setup stub and have the switch happen "in between" packets, so that from remote protocol's point of view, nothing actually happened.*

'`B` *addr*`,`*mode*'

Set (*mode* is '`S`') or clear (*mode* is '`C`') a breakpoint at *addr*.

Don't use this packet. Use the '`Z`' and '`z`' packets instead (see [insert breakpoint or watchpoint packet], page 693).

'`bc`' Backward continue. Execute the target system in reverse. No parameter. See Chapter 6 [Reverse Execution], page 97, for more information.

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

'`bs`' Backward single step. Execute one instruction in reverse. No parameter. See Chapter 6 [Reverse Execution], page 97, for more information.

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

'`c` [*addr*]' Continue at *addr*, which is the address to resume. If *addr* is omitted, resume at current address.

This packet is deprecated for multi-threading support. See [vCont packet], page 690.

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

'`C` *sig*[`;`*addr*]'
Continue with signal *sig* (hex signal number). If '`;`*addr*' is omitted, resume at same address.

This packet is deprecated for multi-threading support. See [vCont packet], page 690.

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

'`d`' Toggle debug flag.

Don't use this packet; instead, define a general set packet (see Section E.4 [General Query Packets], page 699).

'`D`'

'`D;`*pid*' The first form of the packet is used to detach GDB from the remote system. It is sent to the remote target before GDB disconnects via the `detach` command.

The second form, including a process ID, is used when multiprocess protocol extensions are enabled (see [multiprocess extensions], page 715), to detach only a specific process. The *pid* is specified as a big-endian hex string.

Reply:

'`OK`' for success

'`E` *NN*' for an error

'`F` *RC*`,`*EE*`,`*CF*`;`*XX*'
A reply from GDB to an '`F`' packet sent by the target. This is part of the File-I/O protocol extension. See Section E.13 [File-I/O Remote Protocol Extension], page 738, for the specification.

'`g`' Read general registers.

Reply:

'*XX...*' Each byte of register data is described by two hex digits. The bytes with the register are transmitted in target byte order. The

size of each register and their position within the 'g' packet are determined by the GDB internal gdbarch functions DEPRECATED_REGISTER_RAW_SIZE and gdbarch_register_name.

When reading registers from a trace frame (see Section 13.2 [Using the Collected Data], page 205), the stub may also return a string of literal 'x''s in place of the register data digits, to indicate that the corresponding register has not been collected, thus its value is unavailable. For example, for an architecture with 4 registers of 4 bytes each, the following reply indicates to GDB that registers 0 and 2 have not been collected, while registers 1 and 3 have been collected, and both have zero value:

```
-> g
<- xxxxxxxx00000000xxxxxxxx00000000
```

'E *NN*' for an error.

'G *XX...*' Write general registers. See [read registers packet], page 686, for a description of the *XX...* data.

Reply:

'OK' for success

'E *NN*' for an error

'H *op thread-id*'

Set thread for subsequent operations ('m', 'M', 'g', 'G', et.al.). Depending on the operation to be performed, *op* should be 'c' for step and continue operations (note that this is deprecated, supporting the 'vCont' command is a better option), and 'g' for other operations. The thread designator *thread-id* has the format and interpretation described in [thread-id syntax], page 684.

Reply:

'OK' for success

'E *NN*' for an error

'i [*addr*[,*nnn*]]'

Step the remote target by a single clock cycle. If ',*nnn*' is present, cycle step *nnn* cycles. If *addr* is present, cycle step starting at that address.

'I' Signal, then cycle step. See [step with signal packet], page 689. See [cycle step packet], page 687.

'k' Kill request.

The exact effect of this packet is not specified.

For a bare-metal target, it may power cycle or reset the target system. For that reason, the 'k' packet has no reply.

For a single-process target, it may kill that process if possible.

A multiple-process target may choose to kill just one process, or all that are under GDB's control. For more precise control, use the vKill packet (see [vKill packet], page 692).

If the target system immediately closes the connection in response to 'k', GDB does not consider the lack of packet acknowledgment to be an error, and assumes the kill was successful.

If connected using *target extended-remote*, and the target does not close the connection in response to a kill request, GDB probes the target state as if a new connection was opened (see [? packet], page 685).

'm *addr*,*length*'
: Read *length* addressable memory units starting at address *addr* (see [addressable memory unit], page 146). Note that *addr* may not be aligned to any particular boundary.

  The stub need not use any particular size or alignment when gathering data from memory for the response; even if *addr* is word-aligned and *length* is a multiple of the word size, the stub is free to use byte accesses, or not. For this reason, this packet may not be suitable for accessing memory-mapped I/O devices.

  Reply:

  'XX...'
  : Memory contents; each byte is transmitted as a two-digit hexadecimal number. The reply may contain fewer addressable memory units than requested if the server was able to read only part of the region of memory.

  'E NN'
  : *NN* is errno

'M *addr*,*length*:*XX*...'
: Write *length* addressable memory units starting at address *addr* (see [addressable memory unit], page 146). The data is given by *XX*...; each byte is transmitted as a two-digit hexadecimal number.

  Reply:

  'OK'
  : for success

  'E NN'
  : for an error (this includes the case where only part of the data was written).

'p *n*'
: Read the value of register *n*; *n* is in hex. See [read registers packet], page 686, for a description of how the returned register value is encoded.

  Reply:

  'XX...'
  : the register's value

  'E NN'
  : for an error

  ''
  : Indicating an unrecognized *query*.

'P *n*...=*r*...'
: Write register *n*... with value *r*.... The register number *n* is in hexadecimal, and *r*... contains two hex digits for each byte in the register (target byte order).

  Reply:

  'OK'
  : for success

'E *NN*' for an error

'q *name params...*'
'Q *name params...*'
General query ('q') and set ('Q'). These packets are described fully in Section E.4 [General Query Packets], page 699.

'r' Reset the entire system.

Don't use this packet; use the 'R' packet instead.

'R *XX*' Restart the program being debugged. The *XX*, while needed, is ignored. This packet is only available in extended mode (see [extended mode], page 685).

The 'R' packet has no reply.

's [*addr*]' Single step, resuming at *addr*. If *addr* is omitted, resume at same address.

This packet is deprecated for multi-threading support. See [vCont packet], page 690.

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

'S *sig*[;*addr*]'
Step with signal. This is analogous to the 'C' packet, but requests a single-step, rather than a normal resumption of execution.

This packet is deprecated for multi-threading support. See [vCont packet], page 690.

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

't *addr*:*PP*,*MM*'
Search backwards starting at address *addr* for a match with pattern *PP* and mask *MM*, both of which are are 4 byte long. There must be at least 3 digits in *addr*.

'T *thread-id*'
Find out if the thread *thread-id* is alive. See [thread-id syntax], page 684.

Reply:

'OK' thread is still alive

'E *NN*' thread is dead

'v' Packets starting with 'v' are identified by a multi-letter name, up to the first ';' or '?' (or the end of the packet).

'vAttach;*pid*'
Attach to a new process with the specified process ID *pid*. The process ID is a hexadecimal integer identifying the process. In all-stop mode, all threads in the attached process are stopped; in non-stop mode, it may be attached without being stopped if that is supported by the target.

This packet is only available in extended mode (see [extended mode], page 685).

Reply:

'E *nn*' for an error

Note: In non-stop mode, a thread is considered running until GDB acknowledges an asynchronous stop notification for it with the '`vStopped`' packet (see Section E.10 [Remote Non-Stop], page 737).

The stub must support '`vCont`' if it reports support for multiprocess extensions (see [multiprocess extensions], page 715).

Reply: See Section E.3 [Stop Reply Packets], page 695, for the reply specifications.

'`vCont?`' Request a list of actions supported by the '`vCont`' packet.

Reply:

'`vCont[;action...]`'
: The '`vCont`' packet is supported. Each *action* is a supported command in the '`vCont`' packet.

'' : The '`vCont`' packet is not supported.

'`vCtrlC`' Interrupt remote target as if a control-C was pressed on the remote terminal. This is the equivalent to reacting to the `^C` ('`\003`', the control-C character) character in all-stop mode while the target is running, except this works in non-stop mode. See [interrupting remote targets], page 734, for more info on the all-stop variant.

Reply:

'`E nn`' for an error

'`OK`' for success

'`vFile:operation:parameter...`'
: Perform a file operation on the target system. For details, see Section E.7 [Host I/O Packets], page 732.

'`vFlashErase:addr,length`'
: Direct the stub to erase *length* bytes of flash starting at *addr*. The region may enclose any number of flash blocks, but its start and end must fall on block boundaries, as indicated by the flash block size appearing in the memory map (see Section E.16 [Memory Map Format], page 752). GDB groups flash memory programming operations together, and sends a '`vFlashDone`' request after each group; the stub is allowed to delay erase operation until the '`vFlashDone`' packet is received.

Reply:

'`OK`' for success

'`E NN`' for an error

'`vFlashWrite:addr:XX...`'
: Direct the stub to write data to flash address *addr*. The data is passed in binary form using the same encoding as for the '`X`' packet (see [Binary Data], page 683). The memory ranges specified by '`vFlashWrite`' packets preceding a '`vFlashDone`' packet must not overlap, and must appear in order of increasing addresses (although '`vFlashErase`' packets for higher addresses may already

'vStopped'
: See Section E.9 [Notification Packets], page 735.

'X *addr*,*length*:*XX...*'
: Write data to memory, where the data is transmitted in binary. Memory is specified by its address *addr* and number of addressable memory units *length* (see [addressable memory unit], page 146); '*XX...*' is binary data (see [Binary Data], page 683).

  Reply:

  'OK' for success

  'E *NN*' for an error

'z *type*,*addr*,*kind*'
'Z *type*,*addr*,*kind*'
: Insert ('Z') or remove ('z') a *type* breakpoint or watchpoint starting at address *address* of kind *kind*.

  Each breakpoint and watchpoint packet *type* is documented separately.

  Implementation notes: A remote target shall return an empty string for an unrecognized breakpoint or watchpoint packet type. A remote target shall support either both or neither of a given 'Ztype...' and 'ztype...' packet pair. To avoid potential problems with duplicate packets, the operations should be implemented in an idempotent way.

'z0,*addr*,*kind*'
'Z0,*addr*,*kind*[;*cond_list*...][;cmds:*persist*,*cmd_list*...]'
: Insert ('Z0') or remove ('z0') a software breakpoint at address *addr* of type *kind*.

  A software breakpoint is implemented by replacing the instruction at *addr* with a software breakpoint or trap instruction. The *kind* is target-specific and typically indicates the size of the breakpoint in bytes that should be inserted. E.g., the ARM and MIPS can insert either a 2 or 4 byte breakpoint. Some architectures have additional meanings for *kind* (see Section E.5 [Architecture-Specific Protocol Details], page 724); if no architecture-specific value is being used, it should be '0'. *kind* is hex-encoded. *cond_list* is an optional list of conditional expressions in bytecode form that should be evaluated on the target's side. These are the conditions that should be taken into consideration when deciding if the breakpoint trigger should be reported back to GDB.

  See also the 'swbreak' stop reason (see [swbreak stop reason], page 696) for how to best report a software breakpoint event to GDB.

  The *cond_list* parameter is comprised of a series of expressions, concatenated without separators. Each expression has the following form:

  'X *len*,*expr*'
  : *len* is the length of the bytecode expression and *expr* is the actual conditional expression in bytecode form.

  The optional *cmd_list* parameter introduces commands that may be run on the target, rather than being reported back to GDB. The parameter starts with

a numeric flag *persist*; if the flag is nonzero, then the breakpoint may remain active and the commands continue to be run even when GDB disconnects from the target. Following this flag is a series of expressions concatenated with no separators. Each expression has the following form:

'`X` *len*`,`*expr*'
: *len* is the length of the bytecode expression and *expr* is the actual commands expression in bytecode form.

Implementation note: It is possible for a target to copy or move code that contains software breakpoints (e.g., when implementing overlays). The behavior of this packet, in the presence of such a target, is not defined.

Reply:

'`OK`' success

'' not supported

'`E` *NN*' for an error

'`z1,`*addr*`,`*kind*'
'`Z1,`*addr*`,`*kind*[`;`*cond_list*`...`][`;cmds:`*persist*`,`*cmd_list*`...`]'
: Insert ('`Z1`') or remove ('`z1`') a hardware breakpoint at address *addr*.

A hardware breakpoint is implemented using a mechanism that is not dependent on being able to modify the target's memory. The *kind*, *cond_list*, and *cmd_list* arguments have the same meaning as in '`Z0`' packets.

Implementation note: A hardware breakpoint is not affected by code movement.

Reply:

'`OK`' success

'' not supported

'`E` *NN*' for an error

'`z2,`*addr*`,`*kind*'
'`Z2,`*addr*`,`*kind*'
: Insert ('`Z2`') or remove ('`z2`') a write watchpoint at *addr*. The number of bytes to watch is specified by *kind*.

Reply:

'`OK`' success

'' not supported

'`E` *NN*' for an error

'`z3,`*addr*`,`*kind*'
'`Z3,`*addr*`,`*kind*'
: Insert ('`Z3`') or remove ('`z3`') a read watchpoint at *addr*. The number of bytes to watch is specified by *kind*.

Reply:

'`OK`' success

''	not supported
'E *NN*'	for an error

'z4,*addr*,*kind*'
'Z4,*addr*,*kind*'

Insert ('Z4') or remove ('z4') an access watchpoint at *addr*. The number of bytes to watch is specified by *kind*.

Reply:

'OK'	success
''	not supported
'E *NN*'	for an error

E.3 Stop Reply Packets

The 'C', 'c', 'S', 's', 'vCont', 'vAttach', 'vRun', 'vStopped', and '?' packets can receive any of the below as a reply. Except for '?' and 'vStopped', that reply is only returned when the target halts. In the below the exact meaning of *signal number* is defined by the header `include/gdb/signals.h` in the GDB source code.

In non-stop mode, the server will simply reply 'OK' to commands such as 'vCont'; any stop will be the subject of a future notification. See Section E.10 [Remote Non-Stop], page 737.

As in the description of request packets, we include spaces in the reply templates for clarity; these are not part of the reply packet's syntax. No GDB stop reply packet uses spaces to separate its components.

'S *AA*' The program received signal number *AA* (a two-digit hexadecimal number). This is equivalent to a 'T' response with no *n*:*r* pairs.

'T *AA n1*:*r1*;*n2*:*r2*;...'

The program received signal number *AA* (a two-digit hexadecimal number). This is equivalent to an 'S' response, except that the '*n*:*r*' pairs can carry values of important registers and other information directly in the stop reply packet, reducing round-trip latency. Single-step and breakpoint traps are reported this way. Each '*n*:*r*' pair is interpreted as follows:

- If *n* is a hexadecimal number, it is a register number, and the corresponding *r* gives that register's value. The data *r* is a series of bytes in target byte order, with each byte given by a two-digit hex number.
- If *n* is 'thread', then *r* is the *thread-id* of the stopped thread, as specified in [thread-id syntax], page 684.
- If *n* is 'core', then *r* is the hexadecimal number of the core on which the stop event was detected.
- If *n* is a recognized *stop reason*, it describes a more specific event that stopped the target. The currently defined stop reasons are listed below. The *aa* should be '05', the trap signal. At most one stop reason should be present.

The second form of the response, including the process ID of the exited process, can be used only when GDB has reported support for multiprocess protocol extensions; see [multiprocess extensions], page 715. Both *AA* and *pid* are formatted as big-endian hex strings.

'`X` *AA*'
'`X` *AA* `;` `process:`*pid*'
: The process terminated with signal *AA*.

  The second form of the response, including the process ID of the terminated process, can be used only when GDB has reported support for multiprocess protocol extensions; see [multiprocess extensions], page 715. Both *AA* and *pid* are formatted as big-endian hex strings.

'`w` *AA* `;` *tid*'
: The thread exited, and *AA* is the exit status. This response should not be sent by default; GDB requests it with the [QThreadEvents], page 707, packet. See also [thread create event], page 697, above. *AA* is formatted as a big-endian hex string.

'`N`'
: There are no resumed threads left in the target. In other words, even though the process is alive, the last resumed thread has exited. For example, say the target process has two threads: thread 1 and thread 2. The client leaves thread 1 stopped, and resumes thread 2, which subsequently exits. At this point, even though the process is still alive, and thus no '`W`' stop reply is sent, no thread is actually executing either. The '`N`' stop reply thus informs the client that it can stop waiting for stop replies. This packet should not be sent by default; older GDB versions did not support it. GDB requests it, by supplying an appropriate '`qSupported`' feature (see [qSupported], page 709). The remote stub must also supply the appropriate '`qSupported`' feature indicating support.

'`O` *XX*`...`'
: '*XX*`...`' is hex encoding of ASCII data, to be written as the program's console output. This can happen at any time while the program is running and the debugger should continue to wait for '`W`', '`T`', etc. This reply is not permitted in non-stop mode.

'`F` *call-id*`,`*parameter*`...`'
: *call-id* is the identifier which says which host system call should be called. This is just the name of the function. Translation into the correct system call is only applicable as it's defined in GDB. See Section E.13 [File-I/O Remote Protocol Extension], page 738, for a list of implemented system calls.

  '*parameter*`...`' is a list of parameters as defined for this very system call.

  The target replies with this packet when it expects GDB to call a host system call on behalf of the target. GDB replies with an appropriate '`F`' packet and keeps up waiting for the next reply packet from the target. The latest '`C`', '`c`', '`S`' or '`s`' action is expected to be continued. See Section E.13 [File-I/O Remote Protocol Extension], page 738, for more details.

E.4 General Query Packets

Packets starting with '`q`' are *general query packets*; packets starting with '`Q`' are *general set packets*. General query and set packets are a semi-unified form for retrieving and sending information to and from the stub.

The initial letter of a query or set packet is followed by a name indicating what sort of thing the packet applies to. For example, GDB may use a '`qSymbol`' packet to exchange symbol definitions with the stub. These packet names follow some conventions:

- The name must not contain commas, colons or semicolons.
- Most GDB query and set packets have a leading upper case letter.
- The names of custom vendor packets should use a company prefix, in lower case, followed by a period. For example, packets designed at the Acme Corporation might begin with '`qacme.foo`' (for querying foos) or '`Qacme.bar`' (for setting bars).

The name of a query or set packet should be separated from any parameters by a '`:`'; the parameters themselves should be separated by '`,`' or '`;`'. Stubs must be careful to match the full packet name, and check for a separator or the end of the packet, in case two packet names share a common prefix. New packets should not begin with '`qC`', '`qP`', or '`qL`'[1].

Like the descriptions of the other packets, each description here has a template showing the packet's overall syntax, followed by an explanation of the packet's meaning. We include spaces in some of the templates for clarity; these are not part of the packet's syntax. No GDB packet uses spaces to separate its components.

Here are the currently defined query and set packets:

'`QAgent:1`'
'`QAgent:0`'
: Turn on or off the agent as a helper to perform some debugging operations delegated from GDB (see [Control Agent], page 649).

'`QAllow:`*op*`:`*val*`...`'
: Specify which operations GDB expects to request of the target, as a semicolon-separated list of operation name and value pairs. Possible values for *op* include '`WriteReg`', '`WriteMem`', '`InsertBreak`', '`InsertTrace`', '`InsertFastTrace`', and '`Stop`'. *val* is either 0, indicating that GDB will not request the operation, or 1, indicating that it may. (The target can then use this to set up its own internals optimally, for instance if the debugger never expects to insert breakpoints, it may not need to install its own trap handler.)

'`qC`'
: Return the current thread ID.

  Reply:

  '`QC` *thread-id*'
  : Where *thread-id* is a thread ID as documented in [thread-id syntax], page 684.

[1] The '`qP`' and '`qL`' packets predate these conventions, and have arguments without any terminator for the packet name; we suspect they are in widespread use in places that are difficult to upgrade. The '`qC`' packet has no arguments, but some existing stubs (e.g. RedBoot) are known to not check for the end of the packet.

This packet is related to the `unset environment` command; see [unset environment], page 37.

'`QEnvironmentReset`'
: On UNIX-like targets, this packet is used to reset the state of environment variables in the remote target before starting the inferior. In this context, reset means unsetting all environment variables that were previously set by the user (i.e., were not initially present in the environment). It is sent to `gdbserver` before the '`QEnvironmentHexEncoded`' (see [QEnvironmentHexEncoded], page 701) and the '`QEnvironmentUnset`' (see [QEnvironmentUnset], page 701) packets.

  This packet is only available in extended mode (see [extended mode], page 685).

  Reply:

  '`OK`' The request succeeded.

  This packet is not probed by default; the remote stub must request it, by supplying an appropriate '`qSupported`' response (see [qSupported], page 709). This should only be done on targets that actually support passing environment variables to the starting inferior.

'`QSetWorkingDir:`[*directory*]'
: This packet is used to inform the remote server of the intended current working directory for programs that are going to be executed.

  The packet is composed by *directory*, an hex encoded representation of the directory that the remote inferior will use as its current working directory. If *directory* is an empty string, the remote server should reset the inferior's current working directory to its original, empty value.

  This packet is only available in extended mode (see [extended mode], page 685).

  Reply:

  '`OK`' The request succeeded.

'`qfThreadInfo`'
'`qsThreadInfo`'
: Obtain a list of all active thread IDs from the target (OS). Since there may be too many active threads to fit into one reply packet, this query works iteratively: it may require more than one query/reply sequence to obtain the entire list of threads. The first query of the sequence will be the '`qfThreadInfo`' query; subsequent queries in the sequence will be the '`qsThreadInfo`' query.

  NOTE: This packet replaces the '`qL`' query (see below).

  Reply:

  '`m` *thread-id*'
  : A single thread ID

  '`m` *thread-id*`,`*thread-id*`...`'
  : a comma-separated list of thread IDs

  '`l`' (lower case letter '`L`') denotes end of list.

In response to each query, the target will reply with a list of one or more thread IDs, separated by commas. GDB will respond to each reply with a request for more thread ids (using the '`qs`' form of the query), until the target responds with '`l`' (lower-case ell, for *last*). Refer to [thread-id syntax], page 684, for the format of the *thread-id* fields.

Note: GDB *will send the* `qfThreadInfo` *query during the initial connection with the remote target, and the very first thread ID mentioned in the reply will be stopped by* GDB *in a subsequent message. Therefore, the stub should ensure that the first thread ID in the* `qfThreadInfo` *reply is suitable for being stopped by* GDB.

'`qGetTLSAddr:`*thread-id*`,`*offset*`,`*lm*'
Fetch the address associated with thread local storage specified by *thread-id*, *offset*, and *lm*.

thread-id is the thread ID associated with the thread for which to fetch the TLS address. See [thread-id syntax], page 684.

offset is the (big endian, hex encoded) offset associated with the thread local variable. (This offset is obtained from the debug information associated with the variable.)

lm is the (big endian, hex encoded) OS/ABI-specific encoding of the load module associated with the thread local storage. For example, a GNU/Linux system will pass the link map address of the shared object associated with the thread local storage under consideration. Other operating environments may choose to represent the load module differently, so the precise meaning of this parameter will vary.

Reply:

'*XX...*' Hex encoded (big endian) bytes representing the address of the thread local storage requested.

'`E` *nn*' An error occurred. The error number *nn* is given as hex digits.

'' An empty reply indicates that '`qGetTLSAddr`' is not supported by the stub.

'`qGetTIBAddr:`*thread-id*'
Fetch address of the Windows OS specific Thread Information Block.

thread-id is the thread ID associated with the thread.

Reply:

'*XX...*' Hex encoded (big endian) bytes representing the linear address of the thread information block.

'`E` *nn*' An error occured. This means that either the thread was not found, or the address could not be retrieved.

'' An empty reply indicates that '`qGetTIBAddr`' is not supported by the stub.

start address is the starting address of the memory range. The address does not have any restriction on alignment or size.

length is the length, in bytes, of the memory range.

type is the type of tag the request wants to fetch. The type is a signed integer.

tag bytes is a sequence of hex encoded uninterpreted bytes which will be interpreted by the target. Each pair of hex digits is interpreted as a single byte.

Reply:

'`OK`' The request was successful and the memory tag granules were modified accordingly.

'`E` *nn*' An error occured. This means that modifying the memory tag granules failed for some reason.

'' An empty reply indicates that '`QMemTags`' is not supported by the stub, although this should not happen given GDB will only send this packet if the stub has advertised support for memory tagging via '`qSupported`'.

'`qOffsets`'

Get section offsets that the target used when relocating the downloaded image.

Reply:

'`Text=`*xxx*`;Data=`*yyy*[`;Bss=`*zzz*]'

Relocate the `Text` section by *xxx* from its original address. Relocate the `Data` section by *yyy* from its original address. If the object file format provides segment information (e.g. ELF '`PT_LOAD`' program headers),

Reply:

'OK' The request succeeded.

'E *nn*' An error occurred. The error number *nn* is given as hex digits.

'' An empty reply indicates that 'QNonStop' is not supported by the stub.

This packet is not probed by default; the remote stub must request it, by supplying an appropriate 'qSupported' response (see [qSupported], page 709). Use of this packet is controlled by the `set non-stop` command; see Section 5.5.2 [Non-Stop Mode], page 91.

'QCatchSyscalls:1 [;*sysno*]...'
'QCatchSyscalls:0'

Enable ('QCatchSyscalls:1') or disable ('QCatchSyscalls:0') catching syscalls from the inferior process.

For 'QCatchSyscalls:1', each listed syscall *sysno* (encoded in hex) should be reported to GDB. If no syscall *sysno* is listed, every system call should be reported.

Note that if a syscall not in the list is reported, GDB will still filter the event according to its own list from all corresponding `catch syscall` commands. However, it is more efficient to only report the requested syscalls.

Multiple 'QCatchSyscalls:1' packets do not combine; any earlier 'QCatchSyscalls:1' list is completely replaced by the new list.

If the inferior process execs, the state of 'QCatchSyscalls' is kept for the new process too. On targets where exec may affect syscall numbers, for example with exec between 32 and 64-bit processes, the client should send a new packet with the new syscall list.

Reply:

'OK' The request succeeded.

'E *nn*' An error occurred. *nn* are hex digits.

'' An empty reply indicates that 'QCatchSyscalls' is not supported by the stub.

Use of this packet is controlled by the `set remote catch-syscalls` command (see Section 20.4 [Remote Configuration], page 309). This packet is not probed by default; the remote stub must request it, by supplying an appropriate 'qSupported' response (see [qSupported], page 709).

'QPassSignals: *signal* [;*signal*]...'

Each listed *signal* should be passed directly to the inferior process. Signals are numbered identically to continue packets and stop replies (see Section E.3 [Stop Reply Packets], page 695). Each *signal* list item should be strictly greater than the previous item. These signals do not need to stop the inferior, or be reported to GDB. All other signals should be reported to GDB. Multiple 'QPassSignals' packets do not combine; any earlier 'QPassSignals' list is completely replaced

set of threads and synchronously waits for the their corresponding stop replies. Without exit events, if one of the threads exits, GDB would hang forever not knowing that it should no longer expect a stop for that same thread. GDB does not enable this feature unless the stub reports that it supports it by including '`QThreadEvents+`' in its '`qSupported`' reply.

Reply:

'`OK`' The request succeeded.

'`E` *nn*' An error occurred. The error number *nn* is given as hex digits.

'' An empty reply indicates that '`QThreadEvents`' is not supported by the stub.

Use of this packet is controlled by the `set remote thread-events` command (see Section 20.4 [Remote Configuration], page 309).

'`qRcmd,`*command*'

command (hex encoded) is passed to the local interpreter for execution. Invalid commands should be reported using the output string. Before the final result packet, the target may also respond with a number of intermediate '`O`*output*' console output packets. *Implementors should note that providing access to a stubs's interpreter may have security implications.*

Reply:

'`OK`' A command response with no output.

'*OUTPUT*' A command response with the hex encoded output string *OUTPUT*.

'`E` *NN*' Indicate a badly formed request.

'' An empty reply indicates that '`qRcmd`' is not recognized.

(Note that the `qRcmd` packet's name is separated from the command by a '`,`', not a '`:`', contrary to the naming conventions above. Please don't use this packet as a model for new packets.)

'`qSearch:memory:`*address*`;`*length*`;`*search-pattern*'

Search *length* bytes at *address* for *search-pattern*. Both *address* and *length* are encoded in hex; *search-pattern* is a sequence of bytes, also hex encoded.

Reply:

'`0`' The pattern was not found.

'`1,address`'

The pattern was found at *address*.

'`E` *NN*' A badly formed request or an error was encountered while searching memory.

'' An empty reply indicates that '`qSearch:memory`' is not recognized.

'`QStartNoAckMode`'

Request that the remote stub disable the normal '`+`'/'`-`' protocol acknowledgments (see Section E.11 [Packet Acknowledgment], page 737).

Reply:

'`OK`' The stub has switched to no-acknowledgment mode. GDB acknowledges this response, but neither the stub nor GDB shall send or expect further '`+`'/'`-`' acknowledgments in the current connection.

'' An empty reply indicates that the stub does not support no-acknowledgment mode.

'`qSupported` [`:`*gdbfeature* [`;`*gdbfeature*]`...`]'

Tell the remote stub about features supported by GDB, and query the stub for features it supports. This packet allows GDB and the remote stub to take advantage of each others' features. '`qSupported`' also consolidates multiple feature probes at startup, to improve GDB performance—a single larger packet performs better than multiple smaller probe packets on high-latency links. Some features may enable behavior which must not be on by default, e.g. because it would confuse older clients or stubs. Other features may describe packets which could be automatically probed for, but are not. These features must be reported before GDB will use them. This "default unsupported" behavior is not appropriate for all packets, but it helps to keep the initial connection time under control with new versions of GDB which support increasing numbers of packets.

Reply:

'*stubfeature* [`;`*stubfeature*]`...`'

The stub supports or does not support each returned *stubfeature*, depending on the form of each *stubfeature* (see below for the possible forms).

'' An empty reply indicates that '`qSupported`' is not recognized, or that no features needed to be reported to GDB.

The allowed forms for each feature (either a *gdbfeature* in the '`qSupported`' packet, or a *stubfeature* in the response) are:

'*name*`=`*value*'

The remote protocol feature *name* is supported, and associated with the specified *value*. The format of *value* depends on the feature, but it must not include a semicolon.

'*name*`+`' The remote protocol feature *name* is supported, and does not need an associated value.

'*name*`-`' The remote protocol feature *name* is not supported.

'*name*`?`' The remote protocol feature *name* may be supported, and GDB should auto-detect support in some other way when it is needed. This form will not be used for *gdbfeature* notifications, but may be used for *stubfeature* responses.

Whenever the stub receives a '`qSupported`' request, the supplied set of GDB features should override any previous request. This allows GDB to put the stub

gdbfeature

‘qXfer:siginfo:write’	No	‘-’	Yes
‘qXfer:threads:read’	No	‘-’	Yes
‘qXfer:traceframe-info:read’	No	‘-’	Yes
‘qXfer:uib:read’	No	‘-’	Yes
‘qXfer:fdpic:read’	No	‘-’	Yes
‘Qbtrace:off’	Yes	‘-’	Yes
‘Qbtrace:bts’	Yes	‘-’	Yes
‘Qbtrace:pt’	Yes	‘-’	Yes
‘Qbtrace-conf:bts:size’	Yes	‘-’	Yes
‘Qbtrace-conf:pt:size’	Yes	‘-’	Yes
‘QNonStop’	No	‘-’	Yes
‘QCatchSyscalls’	No	‘-’	Yes
‘QPassSignals’	No	‘-’	Yes
‘QStartNoAckMode’	No	‘-’	Yes
‘multiprocess’	No	‘-’	No
‘ConditionalBreakpoints’	No	‘-’	No
‘ConditionalTracepoints’	No	‘-’	No
‘ReverseContinue’	No	‘-’	No
‘ReverseStep’	No	‘-’	No
‘TracepointSource’	No	‘-’	No
‘QAgent’	No	‘-’	No
‘QAllow’	No	‘-’	No
‘QDisableRandomization’	No	‘-’	No

'EnableDisableTracepoints'	No	'-'	No
'QTBuffer:size'	No	'-'	No
'tracenz'	No	'-'	No
'BreakpointCommands'	No	'-'	No
'swbreak'	No	'-'	No
'hwbreak'	No	'-'	No
'fork-events'	No	'-'	No
'vfork-events'	No	'-'	No
'exec-events'	No	'-'	No
'QThreadEvents'	No	'-'	No
'no-resumed'	No	'-'	No
'memory-tagging'	No	'-'	No

These are the currently defined stub features, in more detail:

'PacketSize=*bytes*'
: The remote stub can accept packets up to at least *bytes* in length. GDB will send packets up to this size for bulk transfers, and will never send larger packets. This is a limit on the data characters in the packet, including the frame and checksum. There is no trailing NUL byte in a remote protocol packet; if the stub stores packets in a NUL-terminated format, it should allow an extra byte in its buffer for the NUL. If this stub feature is not supported, GDB guesses based on the size of the 'g' packet response.

'qXfer:auxv:read'
: The remote stub understands the 'qXfer:auxv:read' packet (see [qXfer auxiliary vector read], page 719).

'qXfer:btrace:read'
: The remote stub understands the 'qXfer:btrace:read' packet (see [qXfer btrace read], page 719).

'qXfer:btrace-conf:read'
: The remote stub understands the 'qXfer:btrace-conf:read' packet (see [qXfer btrace-conf read], page 720).

‘`qXfer:exec-file:read`’
: The remote stub understands the ‘`qXfer:exec-file:read`’ packet (see [qXfer executable filename read], page 720).

‘`qXfer:features:read`’
: The remote stub understands the ‘`qXfer:features:read`’ packet (see [qXfer target description read], page 720).

‘`qXfer:libraries:read`’
: The remote stub understands the ‘`qXfer:libraries:read`’ packet (see [qXfer library list read], page 720).

‘`qXfer:libraries-svr4:read`’
: The remote stub understands the ‘`qXfer:libraries-svr4:read`’ packet (see [qXfer svr4 library list read], page 720).

‘`augmented-libraries-svr4-read`’
: The remote stub understands the augmented form of the ‘`qXfer:libraries-svr4:read`’ packet (see [qXfer svr4 library list read], page 720).

‘`qXfer:memory-map:read`’
: The remote stub understands the ‘`qXfer:memory-map:read`’ packet (see [qXfer memory map read], page 721).

‘`qXfer:sdata:read`’
: The remote stub understands the ‘`qXfer:sdata:read`’ packet (see [qXfer sdata read], page 721).

‘`qXfer:siginfo:read`’
: The remote stub understands the ‘`qXfer:siginfo:read`’ packet (see [qXfer siginfo read], page 722).

‘`qXfer:siginfo:write`’
: The remote stub understands the ‘`qXfer:siginfo:write`’ packet (see [qXfer siginfo write], page 723).

‘`qXfer:threads:read`’
: The remote stub understands the ‘`qXfer:threads:read`’ packet (see [qXfer threads read], page 722).

‘`qXfer:traceframe-info:read`’
: The remote stub understands the ‘`qXfer:traceframe-info:read`’ packet (see [qXfer traceframe info read], page 722).

‘`qXfer:uib:read`’
: The remote stub understands the ‘`qXfer:uib:read`’ packet (see [qXfer unwind info block], page 722).

‘`qXfer:fdpic:read`’
: The remote stub understands the ‘`qXfer:fdpic:read`’ packet (see [qXfer fdpic loadmap read], page 722).

'`TracepointSource`'
: The remote stub understands the '`QTDPsrc`' packet that supplies the source form of tracepoint definitions.

'`QAgent`'
: The remote stub understands the '`QAgent`' packet.

'`QAllow`'
: The remote stub understands the '`QAllow`' packet.

'`QDisableRandomization`'
: The remote stub understands the '`QDisableRandomization`' packet.

'`StaticTracepoint`'
: The remote stub supports static tracepoints.

'`InstallInTrace`'
: The remote stub supports installing tracepoint in tracing.

'`EnableDisableTracepoints`'
: The remote stub supports the '`QTEnable`' (see [QTEnable], page 728) and '`QTDisable`' (see [QTDisable], page 728) packets that allow tracepoints to be enabled and disabled while a trace experiment is running.

'`QTBuffer:size`'
: The remote stub supports the '`QTBuffer:size`' (see [QTBuffer-size], page 732) packet that allows to change the size of the trace buffer.

'`tracenz`'
: The remote stub supports the '`tracenz`' bytecode for collecting strings. See Section F.2 [Bytecode Descriptions], page 759, for details about the bytecode.

'`BreakpointCommands`'
: The remote stub supports running a breakpoint's command list itself, rather than reporting the hit to GDB.

'`Qbtrace:off`'
: The remote stub understands the '`Qbtrace:off`' packet.

'`Qbtrace:bts`'
: The remote stub understands the '`Qbtrace:bts`' packet.

'`Qbtrace:pt`'
: The remote stub understands the '`Qbtrace:pt`' packet.

'`Qbtrace-conf:bts:size`'
: The remote stub understands the '`Qbtrace-conf:bts:size`' packet.

'`Qbtrace-conf:pt:size`'
: The remote stub understands the '`Qbtrace-conf:pt:size`' packet.

'`swbreak`'
: The remote stub reports the '`swbreak`' stop reason for memory breakpoints.

'`hwbreak`' The remote stub reports the '`hwbreak`' stop reason for hardware breakpoints.

'`fork-events`'
The remote stub reports the '`fork`' stop reason for fork events.

'`vfork-events`'
The remote stub reports the '`vfork`' stop reason for vfork events and vforkdone events.

'`exec-events`'
The remote stub reports the '`exec`' stop reason for exec events.

'`vContSupported`'
The remote stub reports the supported actions in the reply to '`vCont?`' packet.

'`QThreadEvents`'
The remote stub understands the '`QThreadEvents`' packet.

'`no-resumed`'
The remote stub reports the '`N`' stop reply.

'`memory-tagging`'
The remote stub supports and implements the required memory tagging functionality and understands the '`qMemTags`' (see [qMemTags], page 704) and '`QMemTags`' (see [QMemTags], page 704) packets.

For AArch64 GNU/Linux systems, this feature also requires access to the `/proc/`*`pid`*`/smaps` file so memory mapping page flags can be inspected. This is done via the '`vFile`' requests.

'`qSymbol::`'
Notify the target that GDB is prepared to serve symbol lookup requests. Accept requests from the target for the values of symbols.

Reply:

'`OK`' The target does not need to look up any (more) symbols.

'`qSymbol:`*`sym_name`*'
The target requests the value of symbol *sym_name* (hex encoded). GDB may provide the value by using the '`qSymbol:`*`sym_value`*`:`*`sym_name`*' message, described below.

'`qSymbol:`*`sym_value`*`:`*`sym_name`*'
Set the value of *sym_name* to *sym_value*.

sym_name (hex encoded) is the name of a symbol whose value the target has previously requested.

sym_value (hex) is the value for symbol *sym_name*. If GDB cannot supply a value for *sym_name*, then this field will be empty.

Reply:

'`OK`' The target does not need to look up any (more) symbols.

'qSymbol:*sym_name*'
: The target requests the value of a new symbol *sym_name* (hex encoded). GDB will continue to supply the values of symbols (if available), until the target ceases to request them.

'qTBuffer'
'QTBuffer'
'QTDisconnected'
'QTDP'
'QTDPsrc'
'QTDV'
'qTfP'
'qTfV'
'QTFrame'
'qTMinFTPILen'
: See Section E.6 [Tracepoint Packets], page 725.

'qThreadExtraInfo,*thread-id*'
: Obtain from the target OS a printable string description of thread attributes for the thread *thread-id*; see [thread-id syntax], page 684, for the forms of *thread-id*. This string may contain anything that the target OS thinks is interesting for GDB to tell the user about the thread. The string is displayed in GDB's `info threads` display. Some examples of possible thread extra info strings are '`Runnable`', or '`Blocked on Mutex`'.

  Reply:

  '*XX...*'
  : Where '*XX...*' is a hex encoding of ASCII data, comprising the printable string containing the extra information about the thread's attributes.

  (Note that the `qThreadExtraInfo` packet's name is separated from the command by a '`,`', not a '`:`', contrary to the naming conventions above. Please don't use this packet as a model for new packets.)

'QTNotes'
'qTP'
'QTSave'
'qTsP'
'qTsV'
'QTStart'
'QTStop'
'QTEnable'
'QTDisable'
'QTinit'
'QTro'
'qTStatus'
'qTV'
'qTfSTM'
'qTsSTM'
'qTSTMat'
: See Section E.6 [Tracepoint Packets], page 725.

'qXfer:*object*:read:*annex*:*offset*,*length*'
: Read uninterpreted bytes from the target's special data area identified by the keyword *object*. Request *length* bytes starting at *offset* bytes into the data. The content and encoding of *annex* is specific to *object*; it can supply additional details about what data to access.

  Reply:

  'm *data*'
  : Data *data* (see [Binary Data], page 683) has been read from the target. There may be more data at a higher address (although it is permitted to return 'm' even for the last valid block of data, as long as at least one byte of data was read). It is possible for *data* to have fewer bytes than the *length* in the request.

  'l *data*'
  : Data *data* (see [Binary Data], page 683) has been read from the target. There is no more data to be read. It is possible for *data* to have fewer bytes than the *length* in the request.

  'l'
  : The *offset* in the request is at the end of the data. There is no more data to be read.

  'E00'
  : The request was malformed, or *annex* was invalid.

  'E *nn*'
  : The offset was invalid, or there was an error encountered reading the data. The *nn* part is a hex-encoded `errno` value.

  ''
  : An empty reply indicates the *object* string was not recognized by the stub, or that the object does not support reading.

  Here are the specific requests of this form defined so far. All the 'qXfer:*object*:read:...' requests use the same reply formats, listed above.

  'qXfer:auxv:read::*offset*,*length*'
  : Access the target's *auxiliary vector*. See Section 10.17 [OS Information], page 171. Note *annex* must be empty.

    This packet is not probed by default; the remote stub must request it, by supplying an appropriate 'qSupported' response (see [qSupported], page 709).

  'qXfer:btrace:read:*annex*:*offset*,*length*'
  : Return a description of the current branch trace. See Section E.19 [Branch Trace Format], page 754. The annex part of the generic 'qXfer' packet may have one of the following values:

    `all`
    : Returns all available branch trace.

    `new`
    : Returns all available branch trace if the branch trace changed since the last read request.

    `delta`
    : Returns the new branch trace since the last read request. Adds a new block to the end of the trace that begins at zero and ends at the source location of the first branch in the trace buffer. This extra block is used to stitch traces together.

must be empty unless the remote stub indicated it supports the augmented form of this packet by supplying an appropriate '`qSupported`' response (see [qXfer read], page 719, [qSupported], page 709).

This packet is optional for better performance on SVR4 targets. GDB uses memory read packets to read the SVR4 library list otherwise.

This packet is not probed by default; the remote stub must request it, by supplying an appropriate '`qSupported`' response (see [qSupported], page 709).

If the remote stub indicates it supports the augmented form of this packet then the annex part of the generic '`qXfer`' packet may contain a semicolon-separated list of '*`name=value`*' arguments. The currently supported arguments are:

`start=`*`address`*

A hexadecimal number specifying the address of the '`struct link_map`

Reply:

'*nn*' *nn* (hex encoded) is the number of bytes written. This may be fewer bytes than supplied in the request.

'E00' The request was malformed, or *annex* was invalid.

'E *nn*' The offset was invalid, or there was an error encountered writing the data. The *nn* part is a hex-encoded `errno` value.

'' An empty reply indicates the *object* string was not recognized by the stub, or that the object does not support writing.

Here are the specific requests of this form defined so far. All the '`qXfer:`*object*`:write:...`' requests use the same reply formats, listed above.

'`qXfer:siginfo:write::`*offset*`:`*data*`...`'
Write *data* to the extra signal information on the target system. The annex part of the generic '`qXfer`' packet must be empty (see [qXfer write], page 722).

This packet is not probed by default; the remote stub must request it, by supplying an appropriate '`qSupported`' response (see [qSupported], page 709).

'`qXfer:`*object*`:`*operation*`:...`'
Requests of this form may be added in the future. When a stub does not recognize the *object* keyword, or its support for *object* does not recognize the *operation* keyword, the stub must respond with an empty packet.

'`qAttached:`*pid*'
Return an indication of whether the remote server attached to an existing process or created a new process. When the multiprocess protocol extensions are supported (see [multiprocess extensions], page 715), *pid* is an integer in hexadecimal format identifying the target process. Otherwise, GDB will omit the *pid* field and the query packet will be simplified as '`qAttached`'.

This query is used, for example, to know whether the remote process should be detached or killed when a GDB session is ended with the `quit` command.

Reply:

'1' The remote server attached to an existing process.

'0' The remote server created a new process.

'E *NN*' A badly formed request or an error was encountered.

'`Qbtrace:bts`'
Enable branch tracing for the current thread using Branch Trace Store.

Reply:

'OK' Branch tracing has been enabled.

'E.errtext'
A badly formed request or an error was encountered.

- The system call on the host has been finished.

These two states can be distinguished by the target by the value of the returned `errno`. If it's the protocol representation of `EINTR`, the system call hasn't been performed. This is equivalent to the `EINTR` handling on POSIX systems. In any other case, the target may presume that the system call has been finished — successfully or not — and should behave as if the break message arrived right after the system call.

GDB must behave reliably. If the system call has not been called yet, GDB may send the `F` reply immediately, setting `EINTR` as `errno` in the packet. If the system call on the host has been finished before the user requests a break, the full action must be finished by GDB. This requires sending `M` or `X` packets as necessary. The `F` packet may only be sent when either nothing has happened or the full action has been completed.

E.13.6 Console I/O

By default and if not explicitly closed by the target system, the file descriptors 0, 1 and 2 are connected to the GDB console. Output on the GDB console is handled as any other file output operation (`write(1, ...)` or `write(2, ...)`). Console input is handled by GDB so that after the target read request from file descriptor 0 all following typing is buffered until either one of the following conditions is met:

- The user types *`Ctrl-c`*. The behaviour is as explained above, and the `read` system call is treated as finished.
- The user presses `RET`. This is treated as end of input with a trailing newline.
- The user types *`Ctrl-d`*. This is treated as end of input. No trailing character (neither newline nor '`Ctrl-D`') is appended to the input.

If the user has typed more characters than fit in the buffer given to the `read` call, the trailing characters are buffered in GDB until either another `read(0, ...)` is requested by the target, or debugging is stopped at the user's request.

E.13.7 List of Supported Calls

open

Synopsis:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Request: '`Fopen,`*`pathptr/len`*`,`*`flags`*`,`*`mode`*'

flags is the bitwise `OR` of the following values:

`O_CREAT` If the file does not exist it will be created. The host rules apply as far as file ownership and time stamps are concerned.

`O_EXCL` When used with `O_CREAT`, if the file already exists it is an error and open() fails.

`O_TRUNC` If the file already exists and the open mode allows writing (`O_RDWR` or `O_WRONLY` is given) it will be truncated to zero length.

`O_APPEND` The file is opened in append mode.

`EROFS` The file is on a read-only filesystem.

`ENOSPC` The device containing the file has no room for the new directory entry.

`EINTR` The call was interrupted by the user.

unlink

Synopsis:

```
int unlink(const char *pathname);
```

Request: '`Funlink,`*`pathnameptr/len`*'

Return value:
On success, zero is returned. On error, -1 is returned.

Errors:

`EACCES` No access to the file or the path of the file.

`EPERM` The system does not allow unlinking of directories.

`EBUSY` The file *pathname* cannot be unlinked because it's being used by another process.

`EFAULT` *pathnameptr* is an invalid pointer value.

`ENAMETOOLONG`
pathname was too long.

`ENOENT` A directory component in *pathname* does not exist.

`ENOTDIR` A component of the path is not a directory.

`EROFS` The file is on a read-only filesystem.

`EINTR` The call was interrupted by the user.

stat/fstat

Synopsis:

```
int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Request: '`Fstat,`*`pathnameptr/len`*`,`*`bufptr`*'
'`Ffstat,`*`fd`*`,`*`bufptr`*'

Return value:
On success, zero is returned. On error, -1 is returned.

Errors:

`EBADF` *fd* is not a valid open file.

`ENOENT` A directory component in *pathname* does not exist or the path is an empty string.

`ENOTDIR` A component of the path is not a directory.

`EFAULT` *pathnameptr* is an invalid pointer value.

`EACCES` No access to the file or the path of the file.

`ENAMETOOLONG`
pathname was too long.

`EINTR` The call was interrupted by the user.

gettimeofday

Synopsis:

```
int gettimeofday(struct timeval *tv, void *tz);
```

Request: '`Fgettimeofday,`*`tvptr`*`,`*`tzptr`*'

Return value:
On success, 0 is returned, -1 otherwise.

Errors:

`EINVAL` *tz* is a non-NULL pointer.

`EFAULT` *tvptr* and/or *tzptr* is an invalid pointer value.

isatty

Synopsis:

```
int isatty(int fd);
```

Request: '`Fisatty,`*`fd`*'

Return value:
Returns 1 if *fd* refers to the GDB console, 0 otherwise.

Errors:

`EINTR` The call was interrupted by the user.

Note that the `isatty` call is treated as a special case: it returns 1 to the target if the file descriptor is attached to the GDB console, 0 otherwise. Implementing through system calls would require implementing `ioctl` and would be more complex than needed.

system

Synopsis:

```
int system(const char *command);
```

Request: '`Fsystem,`*`commandptr/len`*'

Return value:
If *len* is zero, the return value indicates whether a shell is available. A zero return value indicates a shell is not available. For non-zero *len*, the value returned is -1 on error and the return status of the command otherwise. Only the exit status of the command is returned, which is extracted from the host's `system` return value by calling `WEXITSTATUS(retval)`. In case `/bin/sh` could not be executed, 127 is returned.

Errors:

`EINTR` The call was interrupted by the user.

GDB takes over the full task of calling the necessary host calls to perform the `system` call. The return value of `system` on the host is simplified before it's returned to the target. Any termination signal information from the child process is discarded, and the return value consists entirely of the exit status of the called command.

Due to security concerns, the `system` call is by default refused by GDB. The user has to allow this call explicitly with the `set remote system-call-allowed 1` command.

`set remote system-call-allowed`
: Control whether to allow the `system` calls in the File I/O protocol for the remote target. The default is zero (disabled).

`show remote system-call-allowed`
: Show whether the `system` calls are allowed in the File I/O protocol.

E.13.8 Protocol-specific Representation of Datatypes

Integral Datatypes

The integral datatypes used in the system calls are `int`, `unsigned int`, `long`, `unsigned long`, `mode_t`, and `time_t`.

`int`, `unsigned int`, `mode_t` and `time_t` are implemented as 32 bit values in this protocol.

`long` and `unsigned long` are implemented as 64 bit types.

See [Limits], page 750, for corresponding MIN and MAX values (similar to those in `limits.h`) to allow range checking on host and target.

`time_t` datatypes are defined as seconds since the Epoch.

All integral datatypes transferred as part of a memory read or write of a structured datatype e.g. a `struct stat` have to be given in big endian byte order.

Pointer Values

Pointers to target data are transmitted as they are. An exception is made for pointers to buffers for which the length isn't transmitted as part of the function call, namely strings. Strings are transmitted as a pointer/length pair, both as hex values, e.g.

```
1aaf/12
```

which is a pointer to data of length 18 bytes at position 0x1aaf. The length is defined as the full string length in bytes, including the trailing null byte. For example, the string `"hello world"` at address 0x123456 is transmitted as

```
123456/d
```

Memory Transfer

Structured data which is transferred using a memory read or write (for example, a `struct stat`) is expected to be in a protocol-specific format with all scalar multibyte datatypes being big endian. Translation to this representation needs to be done both by the target before the `F` packet is sent, and by GDB before it transfers memory to the target. Transferred pointers to structured data should point to the already-coerced data at any time.

struct stat

The buffer of type `struct stat` used by the target and GDB is defined as follows:

```
struct stat {
    unsigned int  st_dev;      /* device */
    unsigned int  st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    unsigned int  st_nlink;    /* number of hard links */
    unsigned int  st_uid;      /* user ID of owner */
    unsigned int  st_gid;      /* group ID of owner */
    unsigned int  st_rdev;     /* device type (if inode device) */
    unsigned long st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

The integral datatypes conform to the definitions given in the appropriate section (see [Integral Datatypes], page 747, for details) so this structure is of size 64 bytes.

The values of several fields have a restricted meaning and/or range of values.

`st_dev` A value of 0 represents a file, 1 the console.

`st_ino` No valid meaning for the target. Transmitted unchanged.

`st_mode` Valid mode bits are described in Section E.13.9 [Constants], page 749. Any other bits have currently no meaning for the target.

`st_uid`
`st_gid`
`st_rdev` No valid meaning for the target. Transmitted unchanged.

`st_atime`
`st_mtime`
`st_ctime` These values have a host and file system dependent accuracy. Especially on Windows hosts, the file system may not support exact timing values.

The target gets a `struct stat` of the above representation and is responsible for coercing it to the target representation before continuing.

Note that due to size differences between the host, target, and protocol representations of `struct stat` members, these members could eventually get truncated on the target.

struct timeval

The buffer of type `struct timeval` used by the File-I/O protocol is defined as follows:

```
struct timeval {
    time_t tv_sec;  /* second */
    long   tv_usec; /* microsecond */
};
```

The integral datatypes conform to the definitions given in the appropriate section (see [Integral Datatypes], page 747, for details) so this structure is of size 8 bytes.

E.13.9 Constants

The following values are used for the constants inside of the protocol. GDB and target are responsible for translating these values before and after the call as needed.

Open Flags

All values are given in hexadecimal representation.

```
O_RDONLY        0x0
O_WRONLY        0x1
O_RDWR          0x2
O_APPEND        0x8
O_CREAT       0x200
O_TRUNC       0x400
O_EXCL        0x800
```

mode_t Values

All values are given in octal representation.

```
S_IFREG       0100000
S_IFDIR        040000
S_IRUSR          0400
S_IWUSR          0200
S_IXUSR          0100
S_IRGRP           040
S_IWGRP           020
S_IXGRP           010
S_IROTH            04
S_IWOTH            02
S_IXOTH            01
```

Errno Values

All values are given in decimal representation.

```
EPERM           1
ENOENT          2
EINTR           4
EBADF           9
EACCES         13
EFAULT         14
EBUSY          16
EEXIST         17
ENODEV         19
ENOTDIR        20
EISDIR         21
EINVAL         22
ENFILE         23
EMFILE         24
EFBIG          27
ENOSPC         28
ESPIPE         29
EROFS          30
ENAMETOOLONG   91
EUNKNOWN     9999
```

`EUNKNOWN` is used as a fallback error value if a host system returns any error value not in the list of supported error numbers.

Lseek Flags

```
SEEK_SET      0
SEEK_CUR      1
SEEK_END      2
```

Limits

All values are given in decimal representation.

```
INT_MIN       -2147483648
INT_MAX        2147483647
UINT_MAX       4294967295
LONG_MIN      -9223372036854775808
LONG_MAX       9223372036854775807
ULONG_MAX      18446744073709551615
```

E.13.10 File-I/O Examples

Example sequence of a write call, file descriptor 3, buffer is at target address 0x1234, 6 bytes should be written:

```
<- Fwrite,3,1234,6
request memory read from target
-> m1234,6
<- XXXXXX
return "6 bytes written"
-> F6
```

Example sequence of a read call, file descriptor 3, buffer is at target address 0x1234, 6 bytes should be read:

```
<- Fread,3,1234,6
request memory write to target
-> X1234,6:XXXXXX
return "6 bytes read"
-> F6
```

Example sequence of a read call, call fails on the host due to invalid file descriptor (EBADF):

```
<- Fread,3,1234,6
-> F-1,9
```

Example sequence of a read call, user presses *Ctrl-c* before syscall on host is called:

```
<- Fread,3,1234,6
-> F-1,4,C
<- T02
```

Example sequence of a read call, user presses *Ctrl-c* after syscall on host is called:

```
<- Fread,3,1234,6
-> X1234,6:XXXXXX
<- T02
```

E.14 Library List Format

On some platforms, a dynamic loader (e.g. `ld.so`) runs in the same process as your application to manage libraries. In this case, GDB can use the loader's symbol table and normal memory operations to maintain a list of shared libraries. On other platforms, the operating system manages loaded libraries. GDB can not retrieve the list of currently loaded libraries

through memory operations, so it uses the '`qXfer:libraries:read`' packet (see [qXfer library list read], page 720) instead. The remote stub queries the target's operating system and reports which libraries are loaded.

The '`qXfer:libraries:read`' packet returns an XML document which lists loaded libraries and their offsets. Each library has an associated name and one or more segment or section base addresses, which report where the library was loaded in memory.

For the common case of libraries that are fully linked binaries, the library should have a list of segments. If the target supports dynamic linking of a relocatable object file, its library XML element should instead include a list of allocated sections. The segment or section bases are start addresses, not relocation offsets; they do not depend on the library's link-time base addresses.

GDB must be linked with the Expat library to support XML library lists. See [Expat], page 661.

A simple memory map, with one loaded library relocated by a single offset, looks like this:

```
<library-list>
  <library name="/lib/libc.so.6">
    <segment address="0x10000000"/>
  </library>
</library-list>
```

Another simple memory map, with one loaded library with three allocated sections (.text, .data, .bss), looks like this:

```
<library-list>
  <library name="sharedlib.o">
    <section address="0x10000000"/>
    <section address="0x20000000"/>
    <section address="0x30000000"/>
  </library>
</library-list>
```

The format of a library list is described by this DTD:

```
<!-- library-list: Root element with versioning -->
<!ELEMENT library-list  (library)*>
<!ATTLIST library-list  version CDATA   #FIXED  "1.0">
<!ELEMENT library       (segment*, section*)>
<!ATTLIST library       name    CDATA   #REQUIRED>
<!ELEMENT segment       EMPTY>
<!ATTLIST segment       address CDATA   #REQUIRED>
<!ELEMENT section       EMPTY>
<!ATTLIST section       address CDATA   #REQUIRED>
```

In addition, segments and section descriptors cannot be mixed within a single library element, and you must supply at least one segment or section for each library.

E.15 Library List Format for SVR4 Targets

On SVR4 platforms GDB can use the symbol table of a dynamic loader (e.g. `ld.so`) and normal memory operations to maintain a list of shared libraries. Still a special library list provided by this packet is more efficient for the GDB remote protocol.

The '`qXfer:libraries-svr4:read`' packet returns an XML document which lists loaded libraries and their SVR4 linker parameters. For each library on SVR4 target, the following parameters are reported:

– `name`, the absolute file name from the `l_name` field of `struct link_map`.
– `lm` with address of `struct link_map` used for TLS (Thread Local Storage) access.
– `l_addr`, the displacement as read from the field `l_addr` of `struct link_map`. For prelinked libraries this is not an absolute memory address. It is a displacement of absolute memory address against address the file was prelinked to during the library load.
– `l_ld`, which is memory address of the `PT_DYNAMIC` segment

Additionally the single `main-lm` attribute specifies address of `struct link_map` used for the main executable. This parameter is used for TLS access and its presence is optional.

GDB must be linked with the Expat library to support XML SVR4 library lists. See [Expat], page 661.

A simple memory map, with two loaded libraries (which do not use prelink), looks like this:

```
<library-list-svr4 version="1.0" main-lm="0xe4f8f8">
  <library name="/lib/ld-linux.so.2" lm="0xe4f51c" l_addr="0xe2d000"
           l_ld="0xe4eefc"/>
  <library name="/lib/libc.so.6" lm="0xe4fbe8" l_addr="0x154000"
           l_ld="0x152350"/>
</library-list-svr>
```

The format of an SVR4 library list is described by this DTD:

```
<!-- library-list-svr4: Root element with versioning -->
<!ELEMENT library-list-svr4  (library)*>
<!ATTLIST library-list-svr4  version CDATA   #FIXED  "1.0">
<!ATTLIST library-list-svr4  main-lm CDATA   #IMPLIED>
<!ELEMENT library            EMPTY>
<!ATTLIST library            name    CDATA   #REQUIRED>
<!ATTLIST library            lm      CDATA   #REQUIRED>
<!ATTLIST library            l_addr  CDATA   #REQUIRED>
<!ATTLIST library            l_ld    CDATA   #REQUIRED>
```

E.16 Memory Map Format

To be able to write into flash memory, GDB needs to obtain a memory map from the target. This section describes the format of the memory map.

The memory map is obtained using the '`qXfer:memory-map:read`' (see [qXfer memory map read], page 721) packet and is an XML document that lists memory regions.

GDB must be linked with the Expat library to support XML memory maps. See [Expat], page 661.

The top-level structure of the document is shown below:

```
<?xml version="1.0"?>
<!DOCTYPE memory-map
          PUBLIC "+//IDN gnu.org//DTD GDB Memory Map V1.0//EN"
                 "http://sourceware.org/gdb/gdb-memory-map.dtd">
<memory-map>
    region...
</memory-map>
```

Each region can be either:

- A region of RAM starting at *addr* and extending for *length* bytes from there:

```
<memory type="ram" start="addr" length="length"/>
```

This list is obtained using the '`qXfer:traceframe-info:read`' (see [qXfer traceframe info read], page 722) packet and is an XML document.

GDB must be linked with the Expat library to support XML traceframe info discovery. See [Expat], page 661.

The top-level structure of the document is shown below:

```
<?xml version="1.0"?>
<!DOCTYPE traceframe-info
          PUBLIC "+//IDN gnu.org//DTD GDB Memory Map V1.0//EN"
                 "http://sourceware.org/gdb/gdb-traceframe-info.dtd">
<traceframe-info>
   block...
</traceframe-info>
```

Each traceframe block can be either:

- A region of collected memory starting at *addr* and extending for *length* bytes from there:

  ```
  <memory start="addr" length="length"/>
  ```

- A block indicating trace state variable numbered *number* has been collected:

  ```
  <tvar id="number"/>
  ```

The formal DTD for the traceframe info format is given below:

```
<!ELEMENT traceframe-info  (memory | tvar)* >
<!ATTLIST traceframe-info  version CDATA   #FIXED  "1.0">

<!ELEMENT memory        EMPTY>
<!ATTLIST memory        start   CDATA   #REQUIRED
                        length  CDATA   #REQUIRED>
<!ELEMENT tvar>
<!ATTLIST tvar          id      CDATA   #REQUIRED>
```

E.19 Branch Trace Format

In order to display the branch trace of an inferior thread, GDB needs to obtain the list of branches. This list is represented as list of sequential code blocks that are connected via branches. The code in each block has been executed sequentially.

This list is obtained using the '`qXfer:btrace:read`' (see [qXfer btrace read], page 719) packet and is an XML document.

GDB must be linked with the Expat library to support XML traceframe info discovery. See [Expat], page 661.

The top-level structure of the document is shown below:

```
<?xml version="1.0"?>
<!DOCTYPE btrace
          PUBLIC "+//IDN gnu.org//DTD GDB Branch Trace V1.0//EN"
                 "http://sourceware.org/gdb/gdb-btrace.dtd">
<btrace>
   block...
</btrace>
```

- A block of sequentially executed instructions starting at *begin* and ending at *end*:

  ```
  <block begin="begin" end="end"/>
  ```

The formal DTD for the branch trace format is given below:

```
<!ELEMENT btrace  (block* | pt) >
<!ATTLIST btrace  version CDATA   #FIXED "1.0">

<!ELEMENT block        EMPTY>
<!ATTLIST block        begin  CDATA   #REQUIRED
                       end    CDATA   #REQUIRED>

<!ELEMENT pt (pt-config?, raw?)>

<!ELEMENT pt-config (cpu?)>

<!ELEMENT cpu EMPTY>
<!ATTLIST cpu vendor   CDATA #REQUIRED
              family   CDATA #REQUIRED
              model    CDATA #REQUIRED
              stepping CDATA #REQUIRED>

<!ELEMENT raw (#PCDATA)>
```

E.20 Branch Trace Configuration Format

For each inferior thread, GDB can obtain the branch trace configuration using the '`qXfer:btrace-conf:read`' (see [qXfer btrace-conf read], page 720) packet.

The configuration describes the branch trace format and configuration settings for that format. The following information is described:

`bts` This thread uses the *Branch Trace Store* (BTS) format.

`size` The size of the BTS ring buffer in bytes.

`pt` This thread uses the *Intel Processor Trace* (Intel PT) format.

`size` The size of the Intel PT ring buffer in bytes.

GDB must be linked with the Expat library to support XML branch trace configuration discovery. See [Expat], page 661.

The formal DTD for the branch trace configuration format is given below:

```
<!ELEMENT btrace-conf (bts?, pt?)>
<!ATTLIST btrace-conf version CDATA #FIXED "1.0">

<!ELEMENT bts EMPTY>
<!ATTLIST bts size CDATA #IMPLIED>

<!ELEMENT pt EMPTY>
<!ATTLIST pt size CDATA #IMPLIED>
```

Appendix F The GDB Agent Expression Mechanism

In some applications, it is not feasible for the debugger to interrupt the program's execution long enough for the developer to learn anything helpful about its behavior. If the program's correctness depends on its real-time behavior, delays introduced by a debugger might cause the program to fail, even when the code itself is correct. It is useful to be able to observe the program's behavior without interrupting it.

Using GDB's `trace` and `collect` commands, the user can specify locations in the program, and arbitrary expressions to evaluate when those locations are reached. Later, using the `tfind` command, she can examine the values those expressions had when the program hit the trace points. The expressions may also denote objects in memory — structures or arrays, for example — whose values GDB should record; while visiting a particular tracepoint, the user may inspect those objects as if they were in memory at that moment. However, because GDB records these values without interacting with the user, it can do so quickly and unobtrusively, hopefully not disturbing the program's behavior.

When GDB is debugging a remote target, the GDB *agent* code running on the target computes the values of the expressions itself. To avoid having a full symbolic expression evaluator on the agent, GDB translates expressions in the source language into a simpler bytecode language, and then sends the bytecode to the agent; the agent then executes the bytecode, and records the values for GDB to retrieve later.

The bytecode language is simple; there are forty-odd opcodes, the bulk of which are the usual vocabulary of C operands (addition, subtraction, shifts, and so on) and various sizes of literals and memory reference operations. The bytecode interpreter operates strictly on machine-level values — various sizes of integers and floating point numbers — and requires no information about types or symbols; thus, the interpreter's internal data structures are simple, and each bytecode requires only a few native machine instructions to implement it. The interpreter is small, and strict limits on the memory and time required to evaluate an expression are easy to determine, making it suitable for use by the debugging agent in real-time applications.

F.1 General Bytecode Design

The agent represents bytecode expressions as an array of bytes. Each instruction is one byte long (thus the term *bytecode*). Some instructions are followed by operand bytes; for example, the `goto` instruction is followed by a destination for the jump.

The bytecode interpreter is a stack-based machine; most instructions pop their operands off the stack, perform some operation, and push the result back on the stack for the next instruction to consume. Each element of the stack may contain either a integer or a floating point value; these values are as many bits wide as the largest integer that can be directly manipulated in the source language. Stack elements carry no record of their type; bytecode could push a value as an integer, then pop it as a floating point value. However, GDB will not generate code which does this. In C, one might define the type of a stack element as follows:

```
union agent_val {
  LONGEST l;
  DOUBLEST d;
```

```
};
```

where `LONGEST` and `DOUBLEST` are `typedef` names for the largest integer and floating point types on the machine.

By the time the bytecode interpreter reaches the end of the expression, the value of the expression should be the only value left on the stack. For tracing applications, `trace` bytecodes in the expression will have recorded the necessary data, and the value on the stack may be discarded. For other applications, like conditional breakpoints, the value may be useful.

Separate from the stack, the interpreter has two registers:

`pc` The address of the next bytecode to execute.

`start` The address of the start of the bytecode expression, necessary for interpreting the `goto` and `if_goto` instructions.

Neither of these registers is directly visible to the bytecode language itself, but they are useful for defining the meanings of the bytecode operations.

There are no instructions to perform side effects on the running program, or call the program's functions; we assume that these expressions are only used for unobtrusive debugging, not for patching the running code.

Most bytecode instructions do not distinguish between the various sizes of values, and operate on full-width values; the upper bits of the values are simply ignored, since they do not usually make a difference to the value computed. The exceptions to this rule are:

memory reference instructions (`ref`*n*)
: There are distinct instructions to fetch different word sizes from memory. Once on the stack, however, the values are treated as full-size integers. They may need to be sign-extended; the `ext` instruction exists for this purpose.

the sign-extension instruction (`ext` *n*)
: These clearly need to know which portion of their operand is to be extended to occupy the full length of the word.

If the interpreter is unable to evaluate an expression completely for some reason (a memory location is inaccessible, or a divisor is zero, for example), we say that interpretation "terminates with an error". This means that the problem is reported back to the interpreter's caller in some helpful way. In general, code using agent expressions should assume that they may attempt to divide by zero, fetch arbitrary memory locations, and misbehave in other ways.

Even complicated C expressions compile to a few bytecode instructions; for example, the expression `x + y * z` would typically produce code like the following, assuming that `x` and `y` live in registers, and `z` is a global variable holding a 32-bit `int`:

```
reg 1
reg 2
const32 address of z
ref32
ext 32
mul
add
```

```
end
```

In detail, these mean:

`reg 1` Push the value of register 1 (presumably holding `x`) onto the stack.

`reg 2` Push the value of register 2 (holding `y`).

`const32` *address of z*
Push the address of `z` onto the stack.

`ref32` Fetch a 32-bit word from the address at the top of the stack; replace the address on the stack with the value. Thus, we replace the address of `z` with `z`'s value.

`ext 32` Sign-extend the value on the top of the stack from 32 bits to full length. This is necessary because `z` is a signed integer.

`mul` Pop the top two numbers on the stack, multiply them, and push their product. Now the top of the stack contains the value of the expression `y * z`.

`add` Pop the top two numbers, add them, and push the sum. Now the top of the stack contains the value of `x + y * z`.

`end` Stop executing; the value left on the stack top is the value to be recorded.

F.2 Bytecode Descriptions

Each bytecode description has the following form:

`add` (0x02): *a b* ⇒ *a*+*b*
Pop the top two stack items, *a* and *b*, as integers; push their sum, as an integer.

In this example, `add` is the name of the bytecode, and `(0x02)` is the one-byte value used to encode the bytecode, in hexadecimal. The phrase "*a b* ⇒ *a*+*b*" shows the stack before and after the bytecode executes. Beforehand, the stack must contain at least two values, *a* and *b*; since the top of the stack is to the right, *b* is on the top of the stack, and *a* is underneath it. After execution, the bytecode will have popped *a* and *b* from the stack, and replaced them with a single value, *a*+*b*. There may be other values on the stack below those shown, but the bytecode affects only those shown.

Here is another example:

`const8` (0x22) *n*: ⇒ *n*
Push the 8-bit integer constant *n* on the stack, without sign extension.

In this example, the bytecode `const8` takes an operand *n* directly from the bytecode stream; the operand follows the `const8` bytecode itself. We write any such operands immediately after the name of the bytecode, before the colon, and describe the exact encoding of the operand in the bytecode stream in the body of the bytecode description.

For the `const8` bytecode, there are no stack items given before the ⇒; this simply means that the bytecode consumes no values from the stack. If a bytecode consumes no values, or produces no values, the list on either side of the ⇒ may be empty.

If a value is written as *a*, *b*, or *n*, then the bytecode treats it as an integer. If a value is written is *addr*, then the bytecode treats it as an address.

`ref_float` (0x1b): *addr* ⇒ *d*
`ref_double` (0x1c): *addr* ⇒ *d*
`ref_long_double` (0x1d): *addr* ⇒ *d*
`l_to_d` (0x1e): *a* ⇒ *d*
`d_to_l` (0x1f): *d* ⇒ *a*
Not implemented yet.

`dup` (0x28): *a* => *a a*
Push another copy of the stack's top element.

`swap` (0x2b): *a b* => *b a*
Exchange the top two items on the stack.

`pop` (0x29): *a* =>
Discard the top value on the stack.

`pick` (0x32) *n*: *a . . . b* => *a . . . b a*
Duplicate an item from the stack and push it on the top of the stack. *n*, a single byte, indicates the stack item to copy. If *n* is zero, this is the same as `dup`; if *n* is one, it copies the item under the top item, etc. If *n* exceeds the number of items on the stack, terminate with an error.

`rot` (0x33): *a b c* => *c a b*
Rotate the top three items on the stack. The top item (c) becomes the third item, the next-to-top item (b) becomes the top item and the third item (a) from the top becomes the next-to-top item.

`if_goto` (0x20) *offset*: *a* ⇒
Pop an integer off the stack; if it is non-zero, branch to the given offset in the bytecode string. Otherwise, continue to the next instruction in the bytecode stream. In other words, if *a* is non-zero, set the `pc` register to `start +` *offset*. Thus, an offset of zero denotes the beginning of the expression.

The *offset* is stored as a sixteen-bit unsigned value, stored immediately following the `if_goto` bytecode. It is always stored most significant byte first, regardless of the target's normal endianness. The offset is not guaranteed to fall at any particular alignment within the bytecode stream; thus, on machines where fetching a 16-bit on an unaligned address raises an exception, you should fetch the offset one byte at a time.

`goto` (0x21) *offset*: ⇒
Branch unconditionally to *offset*; in other words, set the `pc` register to `start +` *offset*.

The offset is stored in the same way as for the `if_goto` bytecode.

`const8` (0x22) *n*: ⇒ *n*
`const16` (0x23) *n*: ⇒ *n*
`const32` (0x24) *n*: ⇒ *n*
`const64` (0x25) *n*: ⇒ *n*
Push the integer constant *n* on the stack, without sign extension. To produce a small negative value, push a small twos-complement value, and then sign-extend it using the `ext` bytecode.

`tracenz` (0x2f) *addr size* ⇒
: Record the bytes at *addr* in a trace buffer, for later retrieval by GDB. Stop at either the first zero byte, or when *size* bytes have been recorded, whichever occurs first.

`printf` (0x34) *numargs string* ⇒
: Do a formatted print, in the style of the C function `printf`). The value of *numargs* is the number of arguments to expect on the stack, while *string* is the format string, prefixed with a two-byte length. The last byte of the string must be zero, and is included in the length. The format string includes escaped sequences just as it appears in C source, so for instance the format string `"\t%d\n"` is six characters long, and the output will consist of a tab character, a decimal number, and a newline. At the top of the stack, above the values to be printed, this bytecode will pop a "function" and "channel". If the function is nonzero, then the target may treat it as a function and call it, passing the channel as a first argument, as with the C function `fprintf`. If the function is zero, then the target may simply call a standard formatted print function of its choice. In all, this bytecode pops 2 + *numargs* stack elements, and pushes nothing.

`end` (0x27): ⇒
: Stop executing bytecode; the result should be the top element of the stack. If the purpose of the expression was to compute an lvalue or a range of memory, then the next-to-top of the stack is the lvalue's address, and the top of the stack is the lvalue's size, in bytes.

F.3 Using Agent Expressions

Agent expressions can be used in several different ways by GDB, and the debugger can generate different bytecode sequences as appropriate.

One possibility is to do expression evaluation on the target rather than the host, such as for the conditional of a conditional tracepoint. In such a case, GDB compiles the source expression into a bytecode sequence that simply gets values from registers or memory, does arithmetic, and returns a result.

Another way to use agent expressions is for tracepoint data collection. GDB generates a different bytecode sequence for collection; in addition to bytecodes that do the calculation, GDB adds `trace` bytecodes to save the pieces of memory that were used.

- The user selects trace points in the program's code at which GDB should collect data.
- The user specifies expressions to evaluate at each trace point. These expressions may denote objects in memory, in which case those objects' contents are recorded as the program runs, or computed values, in which case the values themselves are recorded.
- GDB transmits the tracepoints and their associated expressions to the GDB agent, running on the debugging target.
- The agent arranges to be notified when a trace point is hit.
- When execution on the target reaches a trace point, the agent evaluates the expressions associated with that trace point, and records the resulting values and memory ranges.

- Later, when the user selects a given trace event and inspects the objects and expression values recorded, GDB talks to the agent to retrieve recorded data as necessary to meet the user's requests. If the user asks to see an object whose contents have not been recorded, GDB reports an error.

F.4 Varying Target Capabilities

Some targets don't support floating-point, and some would rather not have to deal with `long long` operations. Also, different targets will have different stack sizes, and different bytecode buffer lengths.

Thus, GDB needs a way to ask the target about itself. We haven't worked out the details yet, but in general, GDB should be able to send the target a packet asking it to describe itself. The reply should be a packet whose length is explicit, so we can add new information to the packet in future revisions of the agent, without confusing old versions of GDB, and it should contain a version number. It should contain at least the following information:

- whether floating point is supported
- whether `long long` is supported
- maximum acceptable size of bytecode stack
- maximum acceptable length of bytecode expressions
- which registers are actually available for collection
- whether the target supports disabled tracepoints

F.5 Rationale

Some of the design decisions apparent above are arguable.

What about stack overflow/underflow?

GDB should be able to query the target to discover its stack size. Given that information, GDB can determine at translation time whether a given expression will overflow the stack. But this spec isn't about what kinds of error-checking GDB ought to do.

Why are you doing everything in LONGEST?

Speed isn't important, but agent code size is; using LONGEST brings in a bunch of support code to do things like division, etc. So this is a serious concern.

First, note that you don't need different bytecodes for different operand sizes. You can generate code without *knowing* how big the stack elements actually are on the target. If the target only supports 32-bit ints, and you don't send any 64-bit bytecodes, everything just works. The observation here is that the MIPS and the Alpha have only fixed-size registers, and you can still get C's semantics even though most instructions only operate on full-sized words. You just need to make sure everything is properly sign-extended at the right times. So there is no need for 32- and 64-bit variants of the bytecodes. Just implement everything using the largest size you support.

GDB should certainly check to see what sizes the target supports, so the user can get an error earlier, rather than later. But this information is not necessary for correctness.

Why don't you have `>` or `<=` operators?

I want to keep the interpreter small, and we don't need them. We can combine the `less_` opcodes with `log_not`, and swap the order of the operands, yielding all four asymmetrical comparison operators. For example, `(x <= y)` is `! (x > y)`, which is `! (y < x)`.

Why do you have `log_not`?
Why do you have `ext`?
Why do you have `zero_ext`?

These are all easily synthesized from other instructions, but I expect them to be used frequently, and they're simple, so I include them to keep bytecode strings short.

`log_not` is equivalent to `const8 0 equal`; it's used in half the relational operators.

`ext` *`n`* is equivalent to `const8` *`s-n`* `lsh const8` *`s-n`* `rsh_signed`, where *s* is the size of the stack elements; it follows `ref`*`m`* and *reg* bytecodes when the value should be signed. See the next bulleted item.

`zero_ext` *`n`* is equivalent to `const`*`m mask`* `log_and`; it's used whenever we push the value of a register, because we can't assume the upper bits of the register aren't garbage.

Why not have sign-extending variants of the `ref` operators?

Because that would double the number of `ref` operators, and we need the `ext` bytecode anyway for accessing bitfields.

Why not have constant-address variants of the `ref` operators?

Because that would double the number of `ref` operators again, and `const32` *`address`* `ref32` is only one byte longer.

Why do the `ref`*`n`* operators have to support unaligned fetches?

GDB will generate bytecode that fetches multi-byte values at unaligned addresses whenever the executable's debugging information tells it to. Furthermore, GDB does not know the value the pointer will have when GDB generates the bytecode, so it cannot determine whether a particular fetch will be aligned or not.

In particular, structure bitfields may be several bytes long, but follow no alignment rules; members of packed structures are not necessarily aligned either.

In general, there are many cases where unaligned references occur in correct C code, either at the programmer's explicit request, or at the compiler's discretion. Thus, it is simpler to make the GDB agent bytecodes work correctly in all circumstances than to make GDB guess in each case whether the compiler did the usual thing.

Why are there no side-effecting operators?

Because our current client doesn't want them? That's a cheap answer. I think the real answer is that I'm afraid of implementing function calls. We should re-visit this issue after the present contract is delivered.

Why aren't the `goto` ops PC-relative?

The interpreter has the base address around anyway for PC bounds checking, and it seemed simpler.

Why is there only one offset size for the `goto` ops?

Offsets are currently sixteen bits. I'm not happy with this situation either:

Suppose we have multiple branch ops with different offset sizes. As I generate code left-to-right, all my jumps are forward jumps (there are no loops in expressions), so I never know the target when I emit the jump opcode. Thus, I have to either always assume the largest offset size, or do jump relaxation on the code after I generate it, which seems like a big waste of time.

I can imagine a reasonable expression being longer than 256 bytes. I can't imagine one being longer than 64k. Thus, we need 16-bit offsets. This kind of reasoning is so bogus, but relaxation is pathetic.

The other approach would be to generate code right-to-left. Then I'd always know my offset size. That might be fun.

Where is the function call bytecode?

When we add side-effects, we should add this.

Why does the `reg` bytecode take a 16-bit register number?

Intel's IA-64 architecture has 128 general-purpose registers, and 128 floating-point registers, and I'm sure it has some random control registers.

Why do we need `trace` and `trace_quick`?

Because GDB needs to record all the memory contents and registers an expression touches. If the user wants to evaluate an expression `x->y->z`, the agent must record the values of `x` and `x->y` as well as the value of `x->y->z`.

Don't the `trace` bytecodes make the interpreter less general?

They do mean that the interpreter contains special-purpose code, but that doesn't mean the interpreter can only be used for that purpose. If an expression doesn't use the `trace` bytecodes, they don't get in its way.

Why doesn't `trace_quick` consume its arguments the way everything else does?

In general, you do want your operators to consume their arguments; it's consistent, and generally reduces the amount of stack rearrangement necessary. However, `trace_quick` is a kludge to save space; it only exists so we needn't write `dup const8` *`SIZE`* `trace` before every memory reference. Therefore, it's okay for it not to consume its arguments; it's meant for a specific context in which we know exactly what it should do with the stack. If we're going to have a kludge, it should be an effective kludge.

Why does `trace16` exist?

That opcode was added by the customer that contracted Cygnus for the data tracing work. I personally think it is unnecessary; objects that large will be quite rare, so it is okay to use `dup const16` *`size`* `trace` in those cases.

Whatever we decide to do with `trace16`, we should at least leave opcode 0x30 reserved, to remain compatible with the customer who added it.

Appendix G Target Descriptions

One of the challenges of using GDB to debug embedded systems is that there are so many minor variants of each processor architecture in use. It is common practice for vendors to start with a standard processor core — ARM, PowerPC, or MIPS, for example — and then make changes to adapt it to a particular market niche. Some architectures have hundreds of variants, available from dozens of vendors. This leads to a number of problems:

- With so many different customized processors, it is difficult for the GDB maintainers to keep up with the changes.
- Since individual variants may have short lifetimes or limited audiences, it may not be worthwhile to carry information about every variant in the GDB source tree.
- When GDB does support the architecture of the embedded system at hand, the task of finding the correct architecture name to give the `set architecture` command can be error-prone.

To address these problems, the GDB remote protocol allows a target system to not only identify itself to GDB, but to actually describe its own features. This lets GDB support processor variants it has never seen before — to the extent that the descriptions are accurate, and that GDB understands them.

GDB must be linked with the Expat library to support XML target descriptions. See [Expat], page 661.

G.1 Retrieving Descriptions

Target descriptions can be read from the target automatically, or specified by the user manually. The default behavior is to read the description from the target. GDB retrieves it via the remote protocol using '`qXfer`' requests (see Section E.4 [General Query Packets], page 699). The *annex* in the '`qXfer`' packet will be '`target.xml`'. The contents of the '`target.xml`' annex are an XML document, of the form described in Section G.2 [Target Description Format], page 769.

Alternatively, you can specify a file to read for the target description. If a file is set, the target will not be queried. The commands to specify a file are:

`set tdesc filename` *path*
: Read the target description from *path*.

`unset tdesc filename`
: Do not read the XML target description from a file. GDB will use the description supplied by the current target.

`show tdesc filename`
: Show the filename to read for a target description, if any.

G.2 Target Description Format

A target description annex is an XML (`http://www.w3.org/XML/`) document which complies with the Document Type Definition provided in the GDB sources in `gdb/features/gdb-target.dtd`. This means you can use generally available tools like `xmllint` to check that your feature descriptions are well-formed and valid. However, to

G.2.3 OS ABI

This optional field was introduced in GDB version 7.0. Previous versions of GDB ignore it.

An '`<osabi>`' element has this form:

```
<osabi>abi-name</osabi>
```

abi-name is an OS ABI name from the same selection accepted by `set osabi` (see Section 22.7 [Configuring the Current ABI], page 352).

G.2.4 Compatible Architecture

This optional field was introduced in GDB version 7.0. Previous versions of GDB ignore it.

A '`<compatible>`' element has this form:

```
<compatible>arch</compatible>
```

arch is one of the architectures from the set accepted by `set architecture` (see Chapter 19 [Specifying a Debugging Target], page 295).

A '`<compatible>`' element is used to specify that the target is able to run binaries in some other than the main target architecture given by the '`<architecture>`' element. For example, on the Cell Broadband Engine, the main architecture is `powerpc:common` or `powerpc:common64`, but the system is able to run binaries in the `spu` architecture as well. The way to describe this capability with '`<compatible>`' is as follows:

```
<architecture>powerpc:common</architecture>
<compatible>spu</compatible>
```

G.2.5 Features

Each '`<feature>`' describes some logical portion of the target system. Features are currently used to describe available CPU registers and the types of their contents. A '`<feature>`' element has this form:

```
<feature name="name">
  [type...]
  reg...
</feature>
```

Each feature's name should be unique within the description. The name of a feature does not matter unless GDB has some special knowledge of the contents of that feature; if it does, the feature should have its standard name. See Section G.5 [Standard Target Features], page 775.

G.2.6 Types

Any register's value is a collection of bits which GDB must interpret. The default interpretation is a two's complement integer, but other types can be requested by name in the register description. Some predefined types are provided by GDB (see Section G.3 [Predefined Target Types], page 773), and the description can define additional composite and enum types.

Each type element must have an '`id`' attribute, which gives a unique (within the containing '`<feature>`') name to the type. Types must be defined before they are used.

Some targets offer vector registers, which can be treated as arrays of scalar elements. These types are written as '`<vector>`' elements, specifying the array element type, *type*, and the number of elements, *count*:

```
<vector id="id" type="type" count="count"/>
```

[type="*type*

`uint8`
`uint16`
`uint24`
`uint32`
`uint64`
`uint128` Unsigned integer types holding the specified number of bits.

`code_ptr`
`data_ptr` Pointers to unspecified code and data. The program counter and any dedicated return address register may be marked as code pointers; printing a code pointer converts it into a symbolic address. The stack pointer and any dedicated address registers may be marked as data pointers.

`ieee_half`
Half precision IEEE floating point.

`ieee_single`
Single precision IEEE floating point.

`ieee_double`
Double precision IEEE floating point.

`bfloat16` The 16-bit *brain floating point* format used e.g. by x86 and ARM.

`arm_fpa_ext`
The 12-byte extended precision format used by ARM FPA registers.

`i387_ext` The 10-byte extended precision format used by x87 registers.

`i386_eflags`
32bit EFLAGS register used by x86.

`i386_mxcsr`
32bit MXCSR register used by x86.

G.4 Enum Target Types

Enum target types are useful in '`struct`' and '`flags`' register descriptions. See Section G.2 [Target Description Format], page 769.

Enum types have a name, size and a list of name/value pairs.

```
<enum id="id" size="size">
  <evalue name="name" value="value"/>
  ...
</enum>
```

Enums must be defined before they are used.

```
<enum id="levels_type" size="4">
  <evalue name="low" value="0"/>
  <evalue name="high" value="1"/>
</enum>
<flags id="flags_type" size="4">
  <field name="X" start="0"/>
  <field name="LEVEL" start="1" end="1" type="levels_type"/>
</flags>
<reg name="flags" bitsize="32" type="flags_type"/>
```

Given that description, a value of 3 for the 'flags' register would be printed as:

```
(gdb) info register flags
flags 0x3 [ X LEVEL=high ]
```

G.5 Standard Target Features

A target description must contain either no registers or all the target's registers. If the description contains no registers, then GDB will assume a default register layout, selected based on the architecture. If the description contains any registers, the default layout will not be used; the standard registers must be described in the target description, in such a way that GDB can recognize them.

This is accomplished by giving specific names to feature elements which contain standard registers. GDB will look for features with those names and verify that they contain the expected registers; if any known feature is missing required registers, or if any required feature is missing, GDB will reject the target description. You can add additional registers to any of the standard features — GDB will display them just as if they were added to an unrecognized feature.

This section lists the known features and their expected contents. Sample XML documents for these features are included in the GDB source tree, in the directory `gdb/features`.

Names recognized by GDB should include the name of the company or organization which selected the name, and the overall architecture to which the feature applies; so e.g. the feature containing ARM core registers is named '`org.gnu.gdb.arm.core`'.

The names of registers are not case sensitive for the purpose of recognizing standard features, but GDB will only display registers using the capitalization used in the description.

G.5.1 AArch64 Features

The '`org.gnu.gdb.aarch64.core`' feature is required for AArch64 targets. It should contain registers '`x0`' through '`x30`', '`sp`', '`pc`', and '`cpsr`'.

The '`org.gnu.gdb.aarch64.fpu`' feature is optional. If present, it should contain registers '`v0`' through '`v31`', '`fpsr`', and '`fpcr`'.

The '`org.gnu.gdb.aarch64.sve`' feature is optional. If present, it should contain registers '`z0`' through '`z31`', '`p0`' through '`p15`', '`ffr`' and '`vg`'.

The '`org.gnu.gdb.aarch64.pauth`' feature is optional. If present, it should contain registers '`pauth_dmask`' and '`pauth_cmask`'.

G.5.2 ARC Features

ARC processors are so configurable that even core registers and their numbers are not predetermined completely. Moreover, *flags* and *PC* registers, which are important to GDB, are not "core" registers in ARC. Therefore, there are two features that their presence is mandatory: '`org.gnu.gdb.arc.core`' and '`org.gnu.gdb.arc.aux`'.

The '`org.gnu.gdb.arc.core`' feature is required for all targets. It must contain registers:

- '`r0`' through '`r25`' for normal register file targets.
- '`r0`' through '`r3`', and '`r10`' through '`r15`' for reduced register file targets.

– ‘gp’, ‘fp’, ‘sp’, ‘r30’[1], ‘blink’, ‘lp_count’, ‘pcl’.

In case of an ARCompact target (ARCv1 ISA), the ‘org.gnu.gdb.arc.core’ feature may contain registers ‘ilink1’ and ‘ilink2’. While in case of ARC EM and ARC HS targets (ARCv2 ISA), register ‘ilink’ may be present. The difference between ARCv1 and ARCv2 is the naming of registers *29th* and *30th*. They are called ‘ilink1’ and ‘ilink2’ for ARCv1 and are optional. For ARCv2, they are called ‘ilink’ and ‘r30’ and only ‘ilink’ is optional. The optionality of ‘ilink*’ registers is because of their inaccessibility during user space debugging sessions.

Extension core registers ‘r32’ through ‘r59’ are optional and their existence depends on the configuration. When debugging GNU/Linux applications, i.e. user space debugging, these core registers are not available.

The ‘org.gnu.gdb.arc.aux’ feature is required for all ARC targets. Here is the list of registers pertinent to this feature:

– mandatory: ‘pc’ and ‘status32’.

– optional: ‘lp_start’, ‘lp_end’, and ‘bta’.

G.5.3 ARM Features

The ‘org.gnu.gdb.arm.core’ feature is required for non-M-profile ARM targets. It should contain registers ‘r0’ through ‘r13’, ‘sp’, ‘lr’, ‘pc’, and ‘cpsr’.

For M-profile targets (e.g. Cortex-M3), the ‘org.gnu.gdb.arm.core’ feature is replaced by ‘org.gnu.gdb.arm.m-profile’. It should contain registers ‘r0’ through ‘r13’, ‘sp’, ‘lr’, ‘pc’, and ‘xpsr’.

The ‘org.gnu.gdb.arm.fpa’ feature is optional. If present, it should contain registers ‘f0’ through ‘f7’ and ‘fps’.

The ‘org.gnu.gdb.arm.m-profile-mve’ feature is optional. If present, it must contain register ‘vpr’.

If the ‘org.gnu.gdb.arm.m-profile-mve’ feature is available, GDB will synthesize the ‘p0’ pseudo register from ‘vpr’ contents.

If the ‘org.gnu.gdb.arm.vfp’ feature is available alongside the ‘org.gnu.gdb.arm.m-profile-mve’ feature, GDB will synthesize the ‘q’ pseudo registers from ‘d’ register contents.

The ‘org.gnu.gdb.xscale.iwmmxt’ feature is optional. If present, it should contain at least registers ‘wR0’ through ‘wR15’ and ‘wCGR0’ through ‘wCGR3’. The ‘wCID’, ‘wCon’, ‘wCSSF’, and ‘wCASF’ registers are optional.

The ‘org.gnu.gdb.arm.vfp’ feature is optional. If present, it should contain at least registers ‘d0’ through ‘d15’. If they are present, ‘d16’ through ‘d31’ should also be included. GDB will synthesize the single-precision registers from halves of the double-precision registers.

The ‘org.gnu.gdb.arm.neon’ feature is optional. It does not need to contain registers; it instructs GDB to display the VFP double-precision registers as vectors and to synthesize the quad-precision registers from pairs of double-precision registers. If this feature is present, ‘org.gnu.gdb.arm.vfp’ must also be present and include 32 double-precision registers.

[1] Not necessary for ARCv1.

G.5.5 LoongArch Features

The '`org.gnu.gdb.loongarch.base`' feature is required for LoongArch targets. It should contain the registers '`r0`' through '`r31`', '`pc`', and '`badv`'. Either the architectural names ('`r0`', '`r1`', etc) can be used, or the ABI names ('`zero`', '`ra`', etc).

G.5.6 MicroBlaze Features

The '`org.gnu.gdb.microblaze.core`' feature is required for MicroBlaze targets. It should contain registers '`r0`' through '`r31`', '`rpc`', '`rmsr`', '`rear`', '`resr`', '`rfsr`', '`rbtr`', '`rpvr`', '`rpvr1`' through '`rpvr11`', '`redr`', '`rpid`', '`rzpr`', '`rtlbx`', '`rtlbsx`', '`rtlblo`', and '`rtlbhi`'.

The '`org.gnu.gdb.microblaze.stack-protect`' feature is optional. If present, it should contain registers '`rshr`' and '`rslr`'

G.5.7 MIPS Features

The '`org.gnu.gdb.mips.cpu`' feature is required for MIPS targets. It should contain registers '`r0`' through '`r31`', '`lo`', '`hi`', and '`pc`'. They may be 32-bit or 64-bit depending on the target.

The '`org.gnu.gdb.mips.cp0`' feature is also required. It should contain at least the '`status`', '`badvaddr`', and '`cause`' registers. They may be 32-bit or 64-bit depending on the target.

The '`org.gnu.gdb.mips.fpu`' feature is currently required, though it may be optional in a future version of GDB. It should contain registers '`f0`' through '`f31`', '`fcsr`', and '`fir`'. They may be 32-bit or 64-bit depending on the target.

The '`org.gnu.gdb.mips.dsp`' feature is optional. It should contain registers '`hi1`' through '`hi3`', '`lo1`' through '`lo3`', and '`dspctl`'. The '`dspctl`' register should be 32-bit and the rest may be 32-bit or 64-bit depending on the target.

The '`org.gnu.gdb.mips.linux`' feature is optional. It should contain a single register, '`restart`', which is used by the Linux kernel to control restartable syscalls.

G.5.8 M68K Features

'`org.gnu.gdb.m68k.core`'
'`org.gnu.gdb.coldfire.core`'
'`org.gnu.gdb.fido.core`'
: One of those features must be always present. The feature that is present determines which flavor of m68k is used. The feature that is present should contain registers '`d0`' through '`d7`', '`a0`' through '`a5`', '`fp`', '`sp`', '`ps`' and '`pc`'.

'`org.gnu.gdb.coldfire.fp`'
: This feature is optional. If present, it should contain registers '`fp0`' through '`fp7`', '`fpcontrol`', '`fpstatus`' and '`fpiaddr`'.

  Note that, despite the fact that this feature's name says '`coldfire`', it is used to describe any floating point registers. The size of the registers must match the main m68k flavor; so, for example, if the primary feature is reported as '`coldfire`', then 64-bit floating point registers are required.

G.5.9 NDS32 Features

The '`org.gnu.gdb.nds32.core`' feature is required for NDS32 targets. It should contain at least registers '`r0`' through '`r10`', '`r15`', '`fp`', '`gp`', '`lp`', '`sp`', and '`pc`'.

The '`org.gnu.gdb.nds32.fpu`' feature is optional. If present, it should contain 64-bit double-precision floating-point registers '`fd0`' through *fdN*, which should be '`fd3`', '`fd7`', '`fd15`', or '`fd31`' based on the FPU configuration implemented.

Note: The first sixteen 64-bit double-precision floating-point registers are overlapped with the thirty-two 32-bit single-precision floating-point registers. The 32-bit single-precision registers, if not being listed explicitly, will be synthesized from halves of the overlapping 64-bit double-precision registers. Listing 32-bit single-precision registers explicitly is deprecated, and the support to it could be totally removed some day.

G.5.10 Nios II Features

The '`org.gnu.gdb.nios2.cpu`' feature is required for Nios II targets. It should contain the 32 core registers ('`zero`', '`at`', '`r2`' through '`r23`', '`et`' through '`ra`'), '`pc`', and the 16 control registers ('`status`' through '`mpuacc`').

G.5.11 Openrisc 1000 Features

The '`org.gnu.gdb.or1k.group0`' feature is required for OpenRISC 1000 targets. It should contain the 32 general purpose registers ('`r0`' through '`r31`'), '`ppc`', '`npc`' and '`sr`'.

G.5.12 PowerPC Features

The '`org.gnu.gdb.power.core`' feature is required for PowerPC targets. It should contain registers '`r0`' through '`r31`', '`pc`', '`msr`', '`cr`', '`lr`', '`ctr`', and '`xer`'. They may be 32-bit or 64-bit depending on the target.

The '`org.gnu.gdb.power.fpu`' feature is optional. It should contain registers '`f0`' through '`f31`' and '`fpscr`'.

The '`org.gnu.gdb.power.altivec`' feature is optional. It should contain registers '`vr0`' through '`vr31`', '`vscr`', and '`vrsave`'. GDB will define pseudo-registers '`v0`' through '`v31`' as aliases for the corresponding '`vrX`' registers.

The '`org.gnu.gdb.power.vsx`' feature is optional. It should contain registers '`vs0h`' through '`vs31h`'. GDB will combine these registers with the floating point registers ('`f0`' through '`f31`') and the altivec registers ('`vr0`' through '`vr31`') to present the 128-bit wide registers '`vs0`' through '`vs63`', the set of vector-scalar registers for POWER7. Therefore, this feature requires both '`org.gnu.gdb.power.fpu`' and '`org.gnu.gdb.power.altivec`'.

The '`org.gnu.gdb.power.spe`' feature is optional. It should contain registers '`ev0h`' through '`ev31h`', '`acc`', and '`spefscr`'. SPE targets should provide 32-bit registers in '`org.gnu.gdb.power.core`' and provide the upper halves in '`ev0h`' through '`ev31h`'. GDB will combine these to present registers '`ev0`' through '`ev31`' to the user.

The '`org.gnu.gdb.power.ppr`' feature is optional. It should contain the 64-bit register '`ppr`'.

The '`org.gnu.gdb.power.dscr`' feature is optional. It should contain the 64-bit register '`dscr`'.

The '`org.gnu.gdb.power.tar`' feature is optional. It should contain the 64-bit register '`tar`'.

The ‘org.gnu.gdb.power.ebb’ feature is optional. It should contain registers ‘bescr’, ‘ebbhr’ and ‘ebbrr’, all 64-bit wide.

The ‘org.gnu.gdb.power.linux.pmu’ feature is optional. It should contain registers ‘mmcr0’, ‘mmcr2’, ‘siar’, ‘sdar’ and ‘sier’, all 64-bit wide. This is the subset of the isa 2.07 server PMU registers provided by GNU/Linux.

The ‘org.gnu.gdb.power.htm.spr’ feature is optional. It should contain registers ‘tfhar’, ‘texasr’ and ‘tfiar’, all 64-bit wide.

The ‘org.gnu.gdb.power.htm.core’ feature is optional. It should contain the checkpointed general-purpose registers ‘cr0’ through ‘cr31’, as well as the checkpointed registers ‘clr’ and ‘cctr’. These registers may all be either 32-bit or 64-bit depending on the target. It should also contain the checkpointed registers ‘ccr’ and ‘cxer’, which should both be 32-bit wide.

The ‘org.gnu.gdb.power.htm.fpu’ feature is optional. It should contain the checkpointed 64-bit floating-point registers ‘cf0’ through ‘cf31’, as well as the checkpointed 64-bit register ‘cfpscr’.

The ‘org.gnu.gdb.power.htm.altivec’ feature is optional. It should contain the checkpointed altivec registers ‘cvr0’ through ‘cvr31’, all 128-bit wide. It should also contain the checkpointed registers ‘cvscr’ and ‘cvrsave’, both 32-bit wide.

The ‘org.gnu.gdb.power.htm.vsx’ feature is optional. It should contain registers ‘cvs0h’ through ‘cvs31h’. GDB will combine these registers with the checkpointed floating point registers (‘cf0’ through ‘cf31’) and the checkpointed altivec registers (‘cvr0’ through ‘cvr31’) to present the 128-bit wide checkpointed vector-scalar registers ‘cvs0’ through ‘cvs63’. Therefore, this feature requires both ‘org.gnu.gdb.power.htm.altivec’ and ‘org.gnu.gdb.power.htm.fpu’.

The ‘org.gnu.gdb.power.htm.ppr’ feature is optional. It should contain the 64-bit checkpointed register ‘cppr’.

The ‘org.gnu.gdb.power.htm.dscr’ feature is optional. It should contain the 64-bit checkpointed register ‘cdscr’.

The ‘org.gnu.gdb.power.htm.tar’ feature is optional. It should contain the 64-bit checkpointed register ‘ctar’.

G.5.13 RISC-V Features

The ‘org.gnu.gdb.riscv.cpu’ feature is required for RISC-V targets. It should contain the registers ‘x0’ through ‘x31’, and ‘pc’. Either the architectural names (‘x0’, ‘x1’, etc) can be used, or the ABI names (‘zero’, ‘ra’, etc).

The ‘org.gnu.gdb.riscv.fpu’ feature is optional. If present, it should contain registers ‘f0’ through ‘f31’, ‘fflags’, ‘frm’, and ‘fcsr’. As with the cpu feature, either the architectural register names, or the ABI names can be used.

The ‘org.gnu.gdb.riscv.virtual’ feature is optional. If present, it should contain registers that are not backed by real registers on the target, but are instead virtual, where the register value is derived from other target state. In many ways these are like GDBs pseudo-registers, except implemented by the target. Currently the only register expected in this set is the one byte ‘priv’ register that contains the target’s privilege level in the least significant two bits.

The '`org.gnu.gdb.riscv.csr`' feature is optional. If present, it should contain all of the target's standard CSRs. Standard CSRs are those defined in the RISC-V specification documents. There is some overlap between this feature and the fpu feature; the '`fflags`', '`frm`', and '`fcsr`' registers could be in either feature. The expectation is that these registers will be in the fpu feature if the target has floating point hardware, but can be moved into the csr feature if the target has the floating point control registers, but no other floating point hardware.

The '`org.gnu.gdb.riscv.vector`' feature is optional. If present, it should contain registers '`v0`' through '`v31`', all of which must be the same size. These requirements are based on the v0.10 draft vector extension, as the vector extension is not yet final. In the event that the register set of the vector extension changes for the final specification, the requirements given here could change for future releases of GDB.

G.5.14 RX Features

The '`org.gnu.gdb.rx.core`' feature is required for RX targets. It should contain the registers '`r0`' through '`r15`', '`usp`', '`isp`', '`psw`', '`pc`', '`intb`', '`bpsw`', '`bpc`', '`fintv`', '`fpsw`', and '`acc`'.

G.5.15 S/390 and System z Features

The '`org.gnu.gdb.s390.core`' feature is required for S/390 and System z targets. It should contain the PSW and the 16 general registers. In particular, System z targets should provide the 64-bit registers '`pswm`', '`pswa`', and '`r0`' through '`r15`'. S/390 targets should provide the 32-bit versions of these registers. A System z target that runs in 31-bit addressing mode should provide 32-bit versions of '`pswm`' and '`pswa`', as well as the general register's upper halves '`r0h`' through '`r15h`', and their lower halves '`r0l`' through '`r15l`'.

The '`org.gnu.gdb.s390.fpr`' feature is required. It should contain the 64-bit registers '`f0`' through '`f15`', and '`fpc`'.

The '`org.gnu.gdb.s390.acr`' feature is required. It should contain the 32-bit registers '`acr0`' through '`acr15`'.

The '`org.gnu.gdb.s390.linux`' feature is optional. It should contain the register '`orig_r2`', which is 64-bit wide on System z targets and 32-bit otherwise. In addition, the feature may contain the '`last_break`' register, whose width depends on the addressing mode, as well as the '`system_call`' register, which is always 32-bit wide.

The '`org.gnu.gdb.s390.tdb`' feature is optional. It should contain the 64-bit registers '`tdb0`', '`tac`', '`tct`', '`atia`', and '`tr0`' through '`tr15`'.

The '`org.gnu.gdb.s390.vx`' feature is optional. It should contain 64-bit wide registers '`v0l`' through '`v15l`', which will be combined by GDB with the floating point registers '`f0`' through '`f15`' to present the 128-bit wide vector registers '`v0`' through '`v15`'. In addition, this feature should contain the 128-bit wide vector registers '`v16`' through '`v31`'.

The '`org.gnu.gdb.s390.gs`' feature is optional. It should contain the 64-bit wide guarded-storage-control registers '`gsd`', '`gssm`', and '`gsepla`'.

The '`org.gnu.gdb.s390.gsbc`' feature is optional. It should contain the 64-bit wide guarded-storage broadcast control registers '`bc_gsd`', '`bc_gssm`', and '`bc_gsepla`'.

G.5.16 Sparc Features

The '`org.gnu.gdb.sparc.cpu`' feature is required for sparc32/sparc64 targets. It should describe the following registers:

- '`g0`' through '`g7`'
- '`o0`' through '`o7`'
- '`l0`' through '`l7`'
- '`i0`' through '`i7`'

They may be 32-bit or 64-bit depending on the target.

Also the '`org.gnu.gdb.sparc.fpu`' feature is required for sparc32/sparc64 targets. It should describe the following registers:

- '`f0`' through '`f31`'
- '`f32`' through '`f62`' for sparc64

The '`org.gnu.gdb.sparc.cp0`' feature is required for sparc32/sparc64 targets. It should describe the following registers:

- '`y`', '`psr`', '`wim`', '`tbr`', '`pc`', '`npc`', '`fsr`', and '`csr`' for sparc32
- '`pc`', '`npc`', '`state`', '`fsr`', '`fprs`', and '`y`' for sparc64

G.5.17 TMS320C6x Features

The '`org.gnu.gdb.tic6x.core`' feature is required for TMS320C6x targets. It should contain registers '`A0`' through '`A15`', registers '`B0`' through '`B15`', '`CSR`' and '`PC`'.

The '`org.gnu.gdb.tic6x.gp`' feature is optional. It should contain registers '`A16`' through '`A31`' and '`B16`' through '`B31`'.

The '`org.gnu.gdb.tic6x.c6xp`' feature is optional. It should contain registers '`TSR`', '`ILC`' and '`RILC`'.

Appendix H Operating System Information

Users of GDB often wish to obtain information about the state of the operating system running on the target—for example the list of processes, or the list of open files. This section describes the mechanism that makes it possible. This mechanism is similar to the target features mechanism (see Appendix G [Target Descriptions], page 769), but focuses on a different aspect of target.

Operating system information is retrieved from the target via the remote protocol, using '`qXfer`' requests (see [qXfer osdata read], page 722). The object name in the request should be '`osdata`', and the *annex* identifies the data to be fetched.

H.1 Process list

When requesting the process list, the *annex* field in the '`qXfer`' request should be '`processes`'. The returned data is an XML document. The formal syntax of this document is defined in `gdb/features/osdata.dtd`.

An example document is:

```
<?xml version="1.0"?>
<!DOCTYPE target SYSTEM "osdata.dtd">
<osdata type="processes">
  <item>
    <column name="pid">1</column>
    <column name="user">root</column>
    <column name="command">/sbin/init</column>
    <column name="cores">1,2,3</column>
  </item>
</osdata>
```

Each item should include a column whose name is '`pid`'. The value of that column should identify the process on the target. The '`user`' and '`command`' columns are optional, and will be displayed by GDB. The '`cores`' column, if present, should contain a comma-separated list of cores that this process is running on. Target may provide additional columns, which GDB currently ignores.

Appendix I Trace File Format

The trace file comes in three parts: a header, a textual description section, and a trace frame section with binary data.

The header has the form `\x7fTRACE0\n`. The first byte is `0x7f` so as to indicate that the file contains binary data, while the `0` is a version number that may have different values in the future.

The description section consists of multiple lines of ASCII text separated by newline characters (`0xa`). The lines may include a variety of optional descriptive or context-setting information, such as tracepoint definitions or register set size. GDB will ignore any line that it does not recognize. An empty line marks the end of this section.

`R` *size*
: Specifies the size of a register block in bytes. This is equal to the size of a `g` packet payload in the remote protocol. *size* is an ascii decimal number. There should be only one such line in a single trace file.

`status` *status*
: Trace status. *status* has the same format as a `qTStatus` remote packet reply. There should be only one such line in a single trace file.

`tp` *payload*
: Tracepoint definition. The *payload* has the same format as `qTfP`/`qTsP` remote packet reply payload. A single tracepoint may take multiple lines of definition, corresponding to the multiple reply packets.

`tsv` *payload*
: Trace state variable definition. The *payload* has the same format as `qTfV`/`qTsV` remote packet reply payload. A single variable may take multiple lines of definition, corresponding to the multiple reply packets.

`tdesc` *payload*
: Target description in XML format. The *payload* is a single line of the XML file. All such lines should be concatenated together to get the original XML file. This file is in the same format as `qXfer features` payload, and corresponds to the main `target.xml` file. Includes are not allowed.

The trace frame section consists of a number of consecutive frames. Each frame begins with a two-byte tracepoint number, followed by a four-byte size giving the amount of data in the frame. The data in the frame consists of a number of blocks, each introduced by a character indicating its type (at least register, memory, and trace state variable). The data in this section is raw binary, not a hexadecimal or other encoding; its endianness matches the target's endianness.

`R` *bytes*
: Register block. The number and ordering of bytes matches that of a `g` packet in the remote protocol. Note that these are the actual bytes, in target order, not a hexadecimal encoding.

`M` *address length bytes...*
: Memory block. This is a contiguous block of memory, at the 8-byte address *address*, with a 2-byte length *length*, followed by *length* bytes.

Each slot in the hash table consists of a pair of `offset_type` values. The first value is the offset of the symbol's name in the constant pool. The second value is the offset of the CU vector in the constant pool.

If both values are 0, then this slot in the hash table is empty. This is ok because while 0 is a valid constant pool index, it cannot be a valid index for both a string and a CU vector.

The hash value for a table entry is computed by applying an iterative hash function to the symbol's name. Starting with an initial value of `r = 0`, each (unsigned) character '`c`' in the string is incorporated into the hash using the formula depending on the index version:

Version 4 The formula is `r = r * 67 + c - 113`.

Versions 5 to 7
The formula is `r = r * 67 + tolower (c) - 113`.

The terminating '`\0`' is not incorporated into the hash.

The step size used in the hash table is computed via `((hash * 17) & (size - 1)) | 1`, where '`hash`' is the hash value, and '`size`' is the size of the hash table. The step size is used to find the next candidate slot when handling a hash collision.

The names of C++ symbols in the hash table are canonicalized. We don't currently have a simple description of the canonicalization algorithm; if you intend to create new index sections, you must read the code.

6. The constant pool. This is simply a bunch of bytes. It is organized so that alignment is correct: CU vectors are stored first, followed by strings.

   A CU vector in the constant pool is a sequence of `offset_type` values. The first value is the number of CU indices in the vector. Each subsequent value is the index and symbol attributes of a CU in the CU list. This element in the hash table is used to indicate which CUs define the symbol and how the symbol is used. See below for the format of each CU index+attributes entry.

   A string in the constant pool is zero-terminated.

Attributes were added to CU index values in `.gdb_index` version 7. If a symbol has multiple uses within a CU then there is one CU index+attributes value for each use.

The format of each CU index+attributes entry is as follows (bit 0 = LSB):

Bits 0-23 This is the index of the CU in the CU list.

Bits 24-27 These bits are reserved for future purposes and must be zero.

Bits 28-30 The kind of the symbol in the CU.

- 0 This value is reserved and should not be used. By reserving zero the full `offset_type` value is backwards compatible with previous versions of the index.
- 1 The symbol is a type.
- 2 The symbol is a variable or an enum value.
- 3 The symbol is a function.

4 Any other kind of symbol.

5,6,7 These values are reserved.

Bit 31 This bit is zero if the value is global and one if it is static.

The determination of whether a symbol is global or static is complicated. The authorative reference is the file `dwarf2read.c` in GDB sources.

This pseudo-code describes the computation of a symbol's kind and global/static attributes in the index.

```
is_external = get_attribute (die, DW_AT_external);
language = get_attribute (cu_die, DW_AT_language);
switch (die->tag)
  {
  case DW_TAG_typedef:
  case DW_TAG_base_type:
  case DW_TAG_subrange_type:
    kind = TYPE;
    is_static = 1;
    break;
  case DW_TAG_enumerator:
    kind = VARIABLE;
    is_static = language != CPLUS;
    break;
  case DW_TAG_subprogram:
    kind = FUNCTION;
    is_static = ! (is_external || language == ADA);
    break;
  case DW_TAG_constant:
    kind = VARIABLE;
    is_static = ! is_external;
    break;
  case DW_TAG_variable:
    kind = VARIABLE;
    is_static = ! is_external;
    break;
  case DW_TAG_namespace:
    kind = TYPE;
    is_static = 0;
    break;
  case DW_TAG_class_type:
  case DW_TAG_interface_type:
  case DW_TAG_structure_type:
  case DW_TAG_union_type:
  case DW_TAG_enumeration_type:
    kind = TYPE;
    is_static = language != CPLUS;
    break;
  default:
    assert (0);
  }
```

Appendix K Download debugging resources with Debuginfod

`debuginfod` is an HTTP server for distributing ELF, DWARF and source files.

With the `debuginfod` client library, `libdebuginfod`, GDB can query servers using the build IDs associated with missing debug info, executables and source files in order to download them on demand.

For instructions on building GDB with `libdebuginfod`, see Section C.5 [–with-debuginfod], page 665. `debuginfod` is packaged with `elfutils`, starting with version 0.178. See `https://sourceware.org/elfutils/Debuginfod.html` for more information regarding `debuginfod`.

K.1 Debuginfod Settings

GDB provides the following commands for configuring `debuginfod`.

`set debuginfod enabled`
`set debuginfod enabled on`
: GDB will attempt to query `debuginfod` servers when missing debug info or source files.

`set debuginfod enabled off`
: GDB will not attempt to query `debuginfod` servers when missing debug info or source files. By default, `debuginfod enabled` is set to `off` for non-interactive sessions.

`set debuginfod enabled ask`
: GDB will prompt the user to enable or disable `debuginfod` before attempting to perform the next query. By default, `debuginfod enabled` is set to `ask` for interactive sessions.

`show debuginfod enabled`
: Display whether `debuginfod enabled` is set to `on`, `off` or `ask`.

`set debuginfod urls`
`set debuginfod urls` *`urls`*
: Set the space-separated list of URLs that `debuginfod` will attempt to query. Only `http://`, `https://` and `file://` protocols should be used. The default value of `debuginfod urls` is copied from the *DEBUGINFOD_URLS* environment variable.

`show debuginfod urls`
: Display the list of URLs that `debuginfod` will attempt to query.

`set debuginfod verbose`
`set debuginfod verbose` *`n`*
: Enable or disable `debuginfod`-related output. Use a non-zero value to enable and `0` to disable. `debuginfod` output is shown by default.

`show debuginfod verbose`
: Show the current verbosity setting.

Appendix L Manual pages

gdb man

gdb [OPTIONS] [*prog*|*prog procID*|*prog core*]

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes – or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C, C++, Fortran and Modula-2.

GDB is invoked with the shell command `gdb`. Once started, it reads commands from the terminal until you tell it to exit with the GDB command `quit` or `exit`. You can get online help from GDB itself by using the command `help`.

You can run `gdb` with no arguments or options; but the most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

```
gdb program
```

You can also start with both an executable program and a core file specified:

```
gdb program core
```

You can, instead, specify a process ID as a second argument or use option `-p`, if you want to debug a running process:

```
gdb program 1234
gdb -p 1234
```

would attach GDB to process `1234`. With option `-p` you can omit the *program* filename.

Here are some of the most frequently needed GDB commands:

`break [`*file*`:][`*function*`|`*line*`]`
: Set a breakpoint at *function* or *line* (in *file*).

`run [`*arglist*`]`
: Start your program (with *arglist*, if specified).

`bt`
: Backtrace: display the program stack.

`print` *expr*
: Display the value of an expression.

`c`
: Continue running your program (after stopping, e.g. at a breakpoint).

`next`
: Execute next program line (after stopping); step *over* any function calls in the line.

`edit [`*file*`:]`*function*
: look at the program line where it is presently stopped.

`list [`*file*`:]`*function*
: type the text of the program in the vicinity of where it is presently stopped.

`step`
: Execute next program line (after stopping); step *into* any function calls in the line.

`help [`*name*`]`
: Show information about GDB command *name*, or general information about using GDB.

`quit`
`exit`
: Exit from GDB.

Any arguments other than options specify an executable file and core file (or process ID); that is, the first argument encountered with no associated option flag is equivalent to a `--se` option, and the second, if any, is equivalent to a `-c` option if it's the name of a file. Many options have both long and abbreviated forms; both are shown here. The long forms are also recognized if you truncate them, so long as enough of the option is present to be unambiguous.

The abbreviated forms are shown here with '`-`' and long forms are shown with '`--`' to reflect how they are shown in `--help`. However, GDB recognizes all of the following conventions for most options:

`--option=`*value*
`--option` *value*
`-option=`*value*
`-option` *value*
`--o=`*value*
`--o` *value*

`-o=`*value*

`-o` *value*

All the options and command line arguments you give are processed in sequential order. The order makes a difference when the `-x` option is used.

`--help`
`-h`
: List all options, with brief explanations.

`--symbols=`*file*
`-s` *file*
: Read symbol table from *file*.

`--write`
: Enable writing into executable and core files.

`--exec=`*file*
`-e` *file*
: Use *file* as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

`--se=`*file*
: Read symbol table from *file* and use it as the executable file.

`--core=`*file*
`-c` *file* Use *file* as a core dump to examine.

`--command=`*file*
`-x` *file* Execute GDB commands from *file*.

`--eval-command=`*command*
`-ex` *command*
Execute given GDB *command*.

`--init-eval-command=`*command*
`-iex` Execute GDB *command* before loading the inferior.

`--directory=`*directory*
`-d` *directory*
Add *directory* to the path to search for source files.

`--nh` Do not execute commands from `~/.config/gdb/gdbinit`, `~/.gdbinit`, `~/.config/gdb/gdbearlyinit`, or `~/.gdbearlyinit`

`--nx`
`-n` Do not execute commands from any `.gdbinit` or `.gdbearlyinit` initialization files.

`--quiet`

`--silent`
`-q` "Quiet". Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

`--batch` Run in batch mode. Exit with status `0` after processing all the command files specified with `-x` (and `.gdbinit`, if not inhibited). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.

Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the message

```
Program exited normally.
```

(which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.

`--batch-silent`
Run in batch mode, just like `--batch`, but totally silent. All GDB output is supressed (stderr is unaffected). This is much quieter than `--silent` and would be useless for an interactive session.

This is particularly useful when using targets that give '`Loading section`' messages, for example.

Note that targets that give their output via GDB, as opposed to writing directly to `stdout`, will also be made silent.

`--args` *prog* [*arglist*]
Change interpretation of command line so that arguments following this option are passed as arguments to the inferior. As an example, take the following command:

```
gdb ./a.out -q
```

It would start GDB with `-q`, not printing the introductory message. On the other hand, using:

```
gdb --args ./a.out -q
```

starts GDB with the introductory message, and passes the option to the inferior.

`--pid=`*pid*
: Attach GDB to an already running program, with the PID *pid*.

`--tui`
: Open the terminal user interface.

`--readnow`
: Read all symbols from the given symfile on the first access.

`--readnever`
: Do not read symbol files.

`--dbx`
: Run in DBX compatibility mode.

`--return-child-result`
: GDB's exit code will be the same as the child's exit code.

`--configuration`
: Print details about GDB configuration and then exit.

`--version`
: Print version information and then exit.

`--cd=`*directory*
: Run GDB using *directory* as its working directory, instead of the current directory.

`--data-directory=`*directory*
`-D`
: Run GDB using *directory* as its data directory. The data directory is where GDB searches for its auxiliary files.

`--fullname`
`-f`
: Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time the program stops). This recognizable format looks like two '`\032`' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two '`\032`' characters as a signal to display the source code for the frame.

`-b` *baudrate*
: Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

`-l` *timeout*
: Set timeout, in seconds, for remote debugging.

`--tty=`*device*
: Run using *device* for your program's standard input and output.

gdbserver man

gdbserver *comm prog* [*args*. . .]

gdbserver –attach *comm pid*

gdbserver –multi *comm*

`gdbserver` is a program that allows you to run GDB on a different machine than the one which is running the program being debugged.

Usage (server (target) side)

First, you need to have a copy of the program you want to debug put onto the target system. The program can be stripped to save space if needed, as `gdbserver` doesn't care about symbols. All symbol handling is taken care of by the GDB running on the host system.

To use the server, you log on to the target system, and run the `gdbserver` program. You must tell it (a) how to communicate with GDB, (b) the name of your program, and (c) its arguments. The general syntax is:

```
target> gdbserver comm program [args ...]
```

For example, using a serial port, you might say:

```
target> gdbserver /dev/com1 emacs foo.txt
```

This tells `gdbserver` to debug emacs with an argument of foo.txt, and to communicate with GDB via `/dev/com1`. `gdbserver` now waits patiently for the host GDB to communicate with it.

To use a TCP connection, you could say:

```
target> gdbserver host:2345 emacs foo.txt
```

This says pretty much the same thing as the last example, except that we are going to communicate with the `host` GDB via TCP. The `host:2345` argument means that we are expecting to see a TCP connection from `host` to local TCP port 2345. (Currently, the `host` part is ignored.) You can choose any number you want for the port number as long as it does not conflict with any existing TCP ports on the target system. This same port number must be used in the host GDBs `target remote` command, which will be described shortly. Note that if you chose a port number that conflicts with another service, `gdbserver` will print an error message and exit.

`gdbserver` can also attach to running programs. This is accomplished via the `--attach` argument. The syntax is:

```
target> gdbserver --attach comm pid
```

pid is the process ID of a currently running process. It isn't necessary to point `gdbserver` at a binary for the running process.

To start `gdbserver` without supplying an initial command to run or process ID to attach, use the `--multi` command line option. In such case you should connect using *`target extended-remote`* to start the program you want to debug.

```
target> gdbserver --multi comm
```

Usage (host side)

You need an unstripped copy of the target program on your host system, since GDB needs to examine its symbol tables and such. Start up GDB as you normally would, with the target program as the first argument. (You may need to use the `--baud` option if the serial line is running at anything except 9600 baud.) That is `gdb TARGET-PROG`, or `gdb --baud BAUD TARGET-PROG`. After that, the only new command you need to know about is `target remote` (or `target extended-remote`). Its argument is either a device name (usually a serial device, like `/dev/ttyb`), or a `HOST:PORT` descriptor. For example:

```
(gdb) target remote /dev/ttyb
```

communicates with the server via serial line `/dev/ttyb`, and:

```
(gdb) target remote the-target:2345
```

communicates via a TCP connection to port 2345 on host 'the-target', where you previously started up `gdbserver` with the same port number. Note that for TCP connections, you must start up `gdbserver` prior to using the 'target remote' command, otherwise you may get an error that looks something like 'Connection refused'.

`gdbserver` can also debug multiple inferiors at once, described in Section 4.9 [Inferiors Connections and Programs], page 40. In such case use the `extended-remote` GDB command variant:

```
(gdb) target extended-remote the-target:2345
```

The `gdbserver` option `--multi` may or may not be used in such case.

There are three different modes for invoking `gdbserver`:

- Debug a specific program specified by its program name:

  ```
  gdbserver comm prog [args...]
  ```

  The *comm* parameter specifies how should the server communicate with GDB; it is either a device name (to use a serial line), a TCP port number (`:1234`), or `-` or `stdio` to use stdin/stdout of `gdbserver`. Specify the name of the program to debug in *prog*. Any remaining arguments will be passed to the program verbatim. When the program exits, GDB will close the connection, and `gdbserver` will exit.

- Debug a specific program by specifying the process ID of a running program:

  ```
  gdbserver --attach comm pid
  ```

  The *comm* parameter is as described above. Supply the process ID of a running program in *pid*; GDB will do everything else. Like with the previous mode, when the process *pid* exits, GDB will close the connection, and `gdbserver` will exit.

- Multi-process mode – debug more than one program/process:

  ```
  gdbserver --multi comm
  ```

  In this mode, GDB can instruct `gdbserver` which command(s) to run. Unlike the other 2 modes, GDB will not close the connection when a process being debugged exits, so you can debug several processes in the same session.

In each of the modes you may specify these options:

`--help` List all options, with brief explanations.

`--version`
This option causes `gdbserver` to print its version number and exit.

`--attach` `gdbserver` will attach to a running program. The syntax is:

```
target> gdbserver --attach comm pid
```

pid is the process ID of a currently running process. It isn't necessary to point `gdbserver` at a binary for the running process.

`--multi` To start `gdbserver` without supplying an initial command to run or process ID to attach, use this command line option. Then you can connect using *`target extended-remote`* and start the program you want to debug. The syntax is:

```
target> gdbserver --multi comm
```

`--debug` Instruct `gdbserver` to display extra status information about the debugging process. This option is intended for `gdbserver` development and for bug reports to the developers.

`--remote-debug`
Instruct `gdbserver` to display remote protocol debug output. This option is intended for `gdbserver` development and for bug reports to the developers.

`--debug-file=`*`filename`*
Instruct `gdbserver` to send any debug output to the given *filename*. This option is intended for `gdbserver` development and for bug reports to the developers.

`--debug-format=option1`[`,option2,...`]
Instruct `gdbserver` to include extra information in each line of debugging output. See [Other Command-Line Arguments for gdbserver], page 306.

`--wrapper`
Specify a wrapper to launch programs for debugging. The option should be followed by the name of the wrapper, then any command-line arguments to pass to the wrapper, then `--` indicating the end of the wrapper arguments.

`--once` By default, `gdbserver` keeps the listening TCP port open, so that additional connections are possible. However, if you start `gdbserver` with the `--once` option, it will stop listening for any further connection attempts after connecting to the first GDB session.

gcore

gcore [-a] [-o *prefix*] *pid1* [*pid2...pidN*]

Generate core dumps of one or more running programs with process IDs *pid1*, *pid2*, etc. A core file produced by `gcore` is equivalent to one produced by the kernel when the process crashes (and when *`ulimit -c`* was used to set up an appropriate core dump limit). However, unlike after a crash, after `gcore` finishes its job the program remains running without any change.

`-a` Dump all memory mappings. The actual effect of this option depends on the Operating System. On GNU/Linux, it will disable `use-coredump-filter` (see [set use-coredump-filter], page 176) and enable `dump-excluded-mappings` (see [set dump-excluded-mappings], page 177).

`-o` *`prefix`* The optional argument *prefix* specifies the prefix to be used when composing the file names of the core dumps. The file name is composed as *`prefix.pid`*,

where *pid* is the process ID of the running program being analyzed by `gcore`. If not specified, *prefix* defaults to *gcore*.

gdbinit

/opt/rh/gcc-toolset-13/root/etc/gdbinit

~/.config/gdb/gdbinit

~/.gdbinit

./.gdbinit

These files contain GDB commands to automatically execute during GDB startup. The lines of contents are canned sequences of commands, described in Section 23.1 [Sequences], page 367.

Please read more in Section 2.1.3 [Startup], page 16.

`/opt/rh/gcc-toolset-13/root/etc/gdbinit`
: System-wide initialization file. It is executed unless user specified GDB option `-nx` or `-n`. See more in

`(not enabled with --with-system-gdbinit-dir during compilation)`
: System-wide initialization directory. All files in this directory are executed on startup unless user specified GDB option `-nx` or `-n`, as long as they have a recognized file extension. See more in Section C.6 [System-wide configuration], page 669.

`~/.config/gdb/gdbinit or ~/.gdbinit`
: User initialization file. It is executed unless user specified GDB options `-nx`, `-n` or `-nh`.

`.gdbinit` Initialization file for current directory. It may need to be enabled with GDB security command `set auto-load local-gdbinit`. See more in Section 22.8.1 [Init File in the Current Directory], page 354.

gdb-add-index

gdb-add-index *filename*

When GDB finds a symbol file, it scans the symbols in the file in order to construct an internal symbol table. This lets most GDB operations work quickly–at the cost of a delay early on. For large programs, this delay can be quite lengthy, so GDB provides a way to build an index, which speeds up startup.

To determine whether a file contains such an index, use the command *`readelf -S filename`*: the index is stored in a section named `.gdb_index`. The index file can only be produced on systems which use ELF binaries and DWARF debug information (i.e., sections named `.debug_*`).

`gdb-add-index` uses GDB and `objdump` found in the `PATH` environment variable. If you want to use different versions of these programs, you can specify them through the `GDB` and `OBJDUMP` environment variables.

See more in Section 18.5 [Index Files], page 291.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

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

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

Appendix N GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

!

'!' packet 685

#

in Modula-2 238

$

$ 161
$$ 161
$_ and info breakpoints 59
$_ and info line 128
$_, $__, and value history 145

&

&, background execution of commands 92

—

--annotate 15
--args 15
--attach, gdbserver option 305
--batch 13
--batch-silent 14
--baud 15
--cd 14
--command 12
--configuration 16
--core 12
--data-directory 14
--debug, gdbserver option 306
--debug-file, gdbserver option 306
--debug-format, gdbserver option 306
--directory 13
--early-init-command 13
--early-init-eval-command 13
--eval-command 12
--exec 12
--fullname 14
--init-command 12
--init-eval-command 12
--interpreter 15
--multi, gdbserver option 299
--nh 13
--nowindows 14
--nx 13
--once, gdbserver option 306
--pid 12
--quiet 13
--readnever, command-line option 13
--readnow 13
--remote-debug, gdbserver option 306
--return-child-result 14
--se 12
--selftest 307
--silent 13
--statistics 16
--symbols 12
--tty 15
--tui 15
--version 16
--windows 14
'--with-gdb-datadir' 294
'--with-relocated-sources' 126
'--with-sysroot' 284
--wrapper, gdbserver option 307
--write 15
-b 15
-c 12
-d 13
-D 14
-e 12
-eiex 13
-eix 13
-ex 12
-f 14
-iex 12
-info-gdb-mi-command 630
-ix 12
-l 15
-n 13
-nw 14
-p 12
-q 13
-r 13
-readnever, option for symbol-file command 278
-s 12
-t 15
-w 14
-x 12

.

., Modula-2 scope operator 238
.build-id directory 286
.debug subdirectories 286
.debug_gdb_scripts section 522
'.gdb_index' section 291
.gdb_index section format 787
.gdbinit 18
'.gnu_debugdata' section 290
.gnu_debuglink sections 288
.note.gnu.build-id sections 288
.o files, reading symbols from 279

/

/proc 321

:

::, context for variables/functions 139

<

<architecture> 770
<compatible> 771
<feature> 771
<flags> 772
<not saved> values 170
<osabi> 771
<reg> 772
<struct> 772
<union> 771
<vector> 771
<xi:include> 770

?

'?' packet 685

_

_NSPrintForDebugger, and printing Objective-C objects 229

'

"No symbol "foo" in current context" 140

{

{*type*} 137

A

'A' packet 685
AArch64 Memory Tagging Extension. 338
AArch64 Pointer Authentication. 338
AArch64 support 338
AArch64 SVE. 338
abbreviation 23
acknowledgment, for GDB remote 737
active targets 295
Ada 238
Ada exception catching 67
Ada exception handlers catching 67
Ada mode, general 238
Ada task switching 245
Ada tasking and core file debugging 246
Ada, deviations from 240
Ada, omissions from 239
Ada, problems 248
Ada, source character set 247
Ada, tasking 243
add new commands for external monitor 304
address locations 122
address of a symbol 252
address size for remote targets 309
addressable memory unit 146
aggregates (Ada) 239
AIX shared library debugging 359
AIX threads 359
aliases for commands 374
aliases for commands, default arguments 376
alignment of remote memory accesses 688
all-stop mode 90
Alpha stack 340
ambiguous expressions 137
annotations 641
annotations for errors, warnings and interrupts 643
annotations for invalidation messages 643
annotations for prompts 642
annotations for running programs 643
annotations for source display 644
append data to a file 175
Application Data Integrity 343
apply a command to all frames (ignoring errors and empty output) 116
apply a command to all frames of all threads (ignoring errors and empty output) 47
apply command to all threads (ignoring errors and empty output) 47
apply command to several frames 114
apply command to several threads 47
ARC EM 332
ARC HS 332
ARC specific commands 332
ARC600 332
ARC700 332
architecture debugging info 359
argument count in user-defined commands 367
arguments (to your program) 36
arguments, to gdbserver 305
arguments, to user-defined commands 367
ARM 32-bit mode 333
ARM AArch64 359
array aggregates (Ada) 239
arrays 141
arrays in expressions 137
arrays slices (Fortran) 230
artificial array 141
assembly instructions 128
assignment 265
async output in GDB/MI 543
async records in GDB/MI 545
asynchronous execution 92, 539
asynchronous execution, and process record and replay 100
AT&T disassembly flavor 131

attach 39
attach to a program, `gdbserver` 305
auto-loading 353
auto-loading extensions 520
auto-loading init file in the current directory .. 354
auto-loading libthread_db.so.1 355
auto-loading safe-path 355
auto-loading verbose mode 357
auto-retry, for remote TCP target 311
automatic display 147
automatic hardware breakpoints 61
automatic overlay debugging 214
automatic symbol index cache 292
automatic thread selection 90
auxiliary vector 171
AVR 337

B

'`b`' packet 685
'`B`' packet 685
background execution 92, 539
backtrace beyond `main` function 110
backtrace limit 111
base name differences 285
baud rate for remote targets 310
'`bc`' packet 685
bcache statistics 677
bits in remote address 309
blocks in guile 502
blocks in python 447
bookmark 52
boundary violations, Intel MPX 89
branch trace configuration format 755
branch trace format 754
branch trace store 100
break in overloaded functions 225
break on a system call. 67
break on fork/exec 67
BREAK signal instead of Ctrl-C 310
breakpoint address adjusted 79
breakpoint at static probe point 121
breakpoint commands 74
breakpoint commands for GDB/MI 554
breakpoint commands, in remote protocol 716
breakpoint conditions 72
breakpoint kinds, ARM 724
breakpoint kinds, MIPS 725
breakpoint lists 55
breakpoint numbers 55
breakpoint on events 55
breakpoint on memory address 55
breakpoint on variable modification 55
breakpoint ranges 55
`breakpoint` subroutine, remote 317
breakpointing Ada elaboration code 242
breakpoints 55
breakpoints and tasks, in Ada 246
breakpoints and threads 93
breakpoints at functions matching a regexp 58
breakpoints in guile 508
breakpoints in overlays 214
breakpoints in python 455
breakpoints, multiple locations 60
'`bs`' packet 686
bug criteria 653
bug reports 653
bugs in GDB 653
build ID sections 288
build ID, and separate debugging files 286
building GDB, requirements for 661
built-in simulator target 296
builtin Go functions 228
builtin Go types 228

C

'`c`' packet 686
C and C++ 221
C and C++ checks 225
C and C++ constants 223
C and C++ defaults 225
C and C++ operators 221
'`C`' packet 686
C++ 221
C++ compilers 224
C++ demangling 226
C++ exception handling 225
C++ overload debugging info 362
C++ scope resolution 140
C++ symbol decoding style 157
C++ symbol display 226
caching data of targets 180
caching of bfd objects 286
caching of opened files 286
call dummy stack unwinding 269
call dummy stack unwinding on unhandled exception. 269
call overloaded functions 224
call stack 107
call stack traces 108
call-clobbered registers 170
caller-saved registers 170
calling functions 269
calling functions in the program, disabling 270
calling make 20
case sensitivity in symbol names 251
case-insensitive symbol names 251
casts, in expressions 137
casts, to view memory 137
catch Ada exceptions 67
catch Ada exceptions when handled 67
catch syscalls from inferior, remote request 706
catchpoints 55
catchpoints, setting 66
change GDB's working directory 38

change inferior's working directory 38
character sets 177
charset 177
checkpoint 52
checkpoints and process id 53
checks, range 220
checks, type 219
checksum, for GDB remote 683
choosing target byte order 298
circular trace buffer 203
clearing breakpoints, watchpoints, catchpoints .. 70
CLI commands in python 429
close, file-i/o system call 743
closest symbol and offset for an address 252
code address and its source line 128
code compression, MIPS 341
COFF/PE exported symbols 360
collected data discarded 202
colon, doubled as scope operator 238
colon-colon, context for variables/functions 139
colors 349
command files 371
command history 346
command hooks 370
command interpreters 525
command line editing 345
command options 27
command options, boolean 27
command options, raw input 27
command scripts, debugging 358
command tracing 358
commands for C++ 225
commands in guile 491
commands in python, CLI 429
commands in python, GDB/MI 433
commands to access guile 469
commands to access python 377
comment 23
common targets 296
COMMON blocks, Fortran 230
compatibility, GDB/MI and CLI 543
compilation directory 124
compile C++ type conversion 273
compile command debugging info 272
compile command driver filename override 276
compile command options override 273
compiling code 271
completion 24
completion of Guile commands 492
completion of Python commands 430
completion of quoted strings 26
completion of structure field names 26
completion of union field names 26
compressed debug sections 662
conditional breakpoints 72
conditional tracepoints 197
configure debuginfod URLs 791
configuring GDB 663
confirmation 358
connection timeout, for remote TCP target 312
connections in python 462
console i/o as part of file-i/o 741
console interpreter 525
console output in GDB/MI 543
constants, in file-i/o protocol 749
continuing 80
continuing threads 89
control C, and remote debugging 317
controlling terminal 38
convenience functions 165
convenience functions in python 438
convenience variables 162
convenience variables for tracepoints 208
convenience variables, and trace
state variables 197
convenience variables, initializing 162
core dump file 277
core dump file target 296
crash of debugger 653
CRC algorithm definition 288
CRC of memory block, remote request 700
CRIS 337
CRIS mode 337
CRIS version 337
Ctrl-BREAK, MS-Windows 326
ctrl-c message, in file-i/o protocol 740
current Ada task ID 244
current directory 124
current Go package 228
current thread 44
current thread, remote request 699
custom JIT debug info 646
Cygwin DLL, debugging 326
Cygwin-specific commands 326

D

'd' packet 686
'D' packet 686
Darwin 331
data breakpoints 55
data manipulation, in GDB/MI 597
dcache line-size 181
dcache size 181
dcache, flushing 181
dead names, GNU Hurd 330
debug expression parser 362
debug formats and C++ 224
debug link sections 288
debug remote protocol 362
debugger crash 653
debugging agent 649
debugging C++ programs 224
debugging information directory, global 286
debugging information in separate files 286
debugging libthread_db 49

debugging multiple processes 50
debugging optimized code 185
debugging stub, example 315
debugging target 295
debugging the Cygwin DLL 326
debugging `threads` 49
debuginfod 791
debuginfod verbosity 791
decimal floating point format 227
default behavior of commands, changing 23
default collection action 200
default data directory 294
default settings, changing 23
default source path substitution 126
default system root 284
define trace state variable, remote request 727
defining macros interactively 189
definition of a macro, showing 189
delete breakpoints 71
deleting breakpoints,
watchpoints, catchpoints 70
deliver a signal to a program 267
demangle 253
demangler crashes 673, 674
demangling C++ names 157
deprecated commands 673
derived type of an object, printing 158
descriptor tables display 324
detach from task, GNU Hurd 330
detach from thread, GNU Hurd 331
direct memory access (DMA) on MS-DOS 325
directories for source files 124
directory, compilation 124
directory, current 124
disable address space
randomization, remote request 700
disabling calling functions in the program 270
disassembler options 131
disconnected tracing 202
displaced stepping debugging info 360
displaced stepping support 672
displaced stepping, and process
record and replay 100
display command history 348
display derived types 158
display disabled out of scope 148
display GDB copyright 30
display of expressions 147
display remote monitor communications 297
display remote packets 362
DJGPP debugging 323
D 228
DLLs with no debugging symbols 327
do not print frame arguments 151
documentation 659
don't repeat command 369
don't repeat Guile command 492
don't repeat Python command 430
DOS file-name semantics of file names 284
DOS serial data link, remote debugging 325
DOS serial port status 325
DPMI 323
dprintf 75
dump all data collected at tracepoint 207
dump core from inferior 176
dump data to a file 175
dump/restore files 175
DVC register 336
DWARF compilation units cache 678
DWARF DIEs 360
DWARF frame unwinders 678
DWARF Line Tables 360
DWARF Reading 360
DWARF-2 CFI and CRIS 337
dynamic linking 279
dynamic printf 75
dynamic varobj 589

E

early initialization 17
early initialization file 16
editing 345
editing source files 123
eight-bit characters in strings 156
elaboration phase 33
ELinOS system-wide configuration script 670
Emacs 535
empty response, for unsupported packets 684
enable debuginfod 791
enable/disable a breakpoint 71
enabling and disabling probes 78
entering numbers 351
environment (of your program) 36
errno values, in file-i/o protocol 749
error on valid input 653
event debugging info 360
event handling 66
event-loop debugging 360
examine process image 321
examining data 133
examining memory 143
exception handlers 66
exceptions, guile 473
exceptions, python 384
exec events, remote reply 697
executable file 277
executable file target 296
executable file, for remote target 311
execute commands from a file 371
execute forward or backward in time 98
execute remote command, remote request 708
execution, foreground, background
and asynchronous 92, 539
exit status of shell commands 165
exiting GDB 19

expand macro once 189
expanding preprocessor macros 189
explicit locations 121
explore type 136
explore value 136
exploring hierarchical data structures 135
expression debugging info 360
expression parser, debugging info 362
expressions 137
expressions in Ada 238
expressions in C or C++ 221
expressions in C++ 224
expressions in Modula-2 232
extend GDB for remote targets 304
extending GDB 367
extra signal information 88

F

'F' packet 686
F reply packet 740
F request packet 740
fast tracepoints 193
fast tracepoints, setting 195
fatal signal 653
fatal signals 86
features of the remote protocol 709
fetch memory tags 704
file name canonicalization 285
file transfer 304
file transfer, remote protocol 732
file-i/o examples 750
file-i/o overview 738
file-i/o reply packet 740
file-i/o request packet 740
File-I/O remote protocol extension 738
filename-display 111
find trace snapshot 205
flinching 358
float promotion 352
floating point 171
floating point registers 169
floating point, MIPS remote 335
focus of debugging 44
foo 293
foreground execution 92, 539
fork events, remote reply 696
fork, debugging programs which call 49
format options 149
formatted output 142
Fortran 1
fortran array slicing debugging info 361
Fortran Defaults 230
Fortran modules, information about 260
Fortran operators and expressions 230
Fortran-specific support in GDB 230
FR-V shared-library debugging 363
frame debugging info 361
frame decorator api 404
frame filters api 402
frame information, printing 154
frame level 107
frame number 107
frame pointer 107
frame pointer register 169
frame, definition 107
frameless execution 107
frames in guile 499
frames in python 443
free memory information (MS-DOS) 324
FreeBSD 332
FreeBSD LWP debug messages 361
FreeBSD native target debug messages 361
fstat, file-i/o system call 745
Fujitsu 316
full symbol tables, listing GDB's internal 261
function call arguments, optimized out 110
function entry/exit, wrong
values of variables 140
functions and variables by Fortran module 260
functions without line info, and stepping 81

G

'g' packet 686
g++, GNU C++ compiler 221
'G' packet 687
garbled pointers 324
GCC and C++ 224
GDB bugs, reporting 653
GDB internal error 674
gdb module 379
gdb objects 472
GDB reference card 659
GDB startup 16
GDB version number 30
gdb.ini 18
gdb.printing 466
gdb.prompt 468
gdb.types 466
gdb.Value 385
GDB/MI development 543
GDB/MI, async records 545
GDB/MI, breakpoint commands 554
GDB/MI, compatibility with CLI 543
GDB/MI, data manipulation 597
GDB/MI, input syntax 540
GDB/MI, its purpose 537
GDB/MI, output syntax 541
GDB/MI, result records 544
GDB/MI, simple examples 553
GDB/MI, stream records 545
GDB/MI General Design 537
gdbarch debugging info 359
GDBHISTFILE, environment variable 346
GDBHISTSIZE, environment variable 347

`gdbinit` ... 18
`gdbserver`, command-line arguments ... 305
`gdbserver`, connecting ... 299
gdbserver, search path for `libthread_db` ... 308
`gdbserver`, send all debug output to a single file ... 306
`gdbserver`, `target extended-remote` mode ... 299
`gdbserver`, `target remote` mode ... 299
`gdbserver`, types of connections ... 299
GDT ... 324
general initialization ... 17
get thread information block address ... 703
get thread-local storage address, remote request ... 703
gettimeofday, file-i/o system call ... 746
getting structure elements using gdb.Field objects as subscripts ... 385
global debugging information directories ... 286
global thread identifier (GDB) ... 45
global thread number ... 45
GNAT descriptive types ... 248
GNAT encoding ... 248
GNU C++ ... 221
GNU Emacs ... 535
GNU Hurd debugging ... 329
GNU/Hurd debug messages ... 361
GNU/Linux namespaces debug messages ... 362
GNU/Linux native target debug messages ... 361
Go (programming language) ... 228
guile api ... 469
guile architectures ... 513
guile auto-loading ... 519
guile commands ... 469, 491
guile configuration ... 471
guile exceptions ... 473
guile gdb module ... 470
guile iterators ... 517
guile modules ... 519
guile pagination ... 470
guile parameters ... 495
guile pretty printing api ... 486
guile scripting ... 468
guile scripts directory ... 469
guile stdout ... 470
guile, working with types ... 482
guile, working with values from inferior ... 475

H

'`H`' packet ... 687
handling signals ... 86
hardware breakpoints ... 57
hardware debug registers ... 679
hardware watchpoints ... 63
hash mark while downloading ... 297
`heuristic-fence-post` (Alpha, MIPS) ... 340
history expansion, turn on/off ... 347
history file ... 346
history number ... 161
history of values printed by GDB ... 161
history size ... 347
history substitution ... 346
hooks, for commands ... 370
hooks, post-command ... 370
hooks, pre-command ... 370
host character set ... 177
Host I/O, remote protocol ... 732
how many arguments (user-defined commands) ... 367
HPPA support ... 342

I

'`i`' packet ... 687
i/o ... 38
i386 ... 316
`i386-stub.c` ... 316
'`I`' packet ... 687
I/O registers (Atmel AVR) ... 337
IDT ... 324
ignore count (of breakpoint) ... 74
in-process agent protocol ... 649
incomplete type ... 256
indentation in structure display ... 156
index files ... 291
index section format ... 787
inferior ... 40
inferior debugging info ... 361
inferior events in Python ... 420
inferior functions, calling ... 269
inferior tty ... 39
inferiors in Python ... 419
infinite recursion in user-defined commands ... 369
info for known .debug_gdb_scripts-loaded scripts ... 677
info for known object files ... 676
info line, repeated calls ... 128
info proc cmdline ... 322
info proc cwd ... 322
info proc exe ... 322
info proc files ... 322
information about static tracepoint markers ... 201
information about tracepoints ... 200
inheritance ... 225
init file ... 16
init file name ... 17
initial frame ... 107
initialization file ... 17
injecting code ... 271
inline functions, debugging ... 185
innermost frame ... 107
input syntax for GDB/MI ... 540
installation ... 661
instructions, assembly ... 128
integral datatypes, in file-i/o protocol ... 747
Intel ... 316

Intel disassembly flavor 131
Intel Memory Protection Extensions (MPX)... 339
Intel MPX boundary violations 89
Intel Processor Trace 100
internal commands 671
internal errors, control of GDB behavior 674
internal GDB breakpoints 62
interrupt 19
interrupt debuggee on MS-Windows 326
interrupt remote programs 310, 311
interrupting remote programs 303
interrupting remote targets 317
interrupts (remote protocol) 734
invalid input 653
invoke another interpreter 525
ipa protocol commands 651
ipa protocol objects 650
isatty, file-i/o system call 746

J

JIT compilation interface 645
JIT debug info reader 646
just-in-time compilation 645
just-in-time compilation,
debugging messages 361

K

'k' packet 687
kernel crash dump 321
kernel memory image 321

L

languages 217
last tracepoint number 196
latest breakpoint 56
lazy strings in guile 512
lazy strings in python 459
LDT 324
leaving GDB 19
libkvm 321
library list format, remote protocol 750, 751
limit hardware breakpoints and watchpoints ... 310
limit hardware watchpoints length 311
limit on number of printed array elements 151
limits, in file-i/o protocol 750
line tables in python 454
line tables, listing GDB's internal 262
linespec locations 120
Linux native targets 361
list active threads, remote request 702
list of supported file-i/o calls 741
list output in GDB/MI 543
list, how many lines to display 119
listing GDB's internal line tables 262
listing GDB's internal symbol tables 261
listing machine instructions 128
listing mapped overlays 213
lists of breakpoints 55
load address, overlay's 211
load shared library 282
load symbols from memory 280
local socket, target remote 301
local variables 257
locate address 142
location 120
lock scheduler 90
log output in GDB/MI 543
logging file name 21
logging GDB output 20
lseek flags, in file-i/o protocol 750
lseek, file-i/o system call 744

M

'm' packet 688
m680x0 316
m68k-stub.c 316
'M' packet 688
Mach-O symbols processing 362
machine instructions 128
macro definition, showing 189
macro expansion, showing the results
of preprocessor 189
macros, example of debugging with 190
macros, from debug info 189
macros, user-defined 189
mailing lists 544
maintenance commands 671
Man pages 793
managing frame filters 116
manual overlay debugging 213
map an overlay 213
mapinfo list, QNX Neutrino 323
mapped address 211
mapped overlays 211
markers, static tracepoints 193
maximum value for offset of closest symbol 150
member functions 224
memory address space mappings 322, 675
memory map format 752
memory region attributes 173
memory tag types, ARM 725
memory tracing 55
memory transfer, in file-i/o protocol 747
memory used by commands 681
memory used for symbol tables 282
memory, alignment and size of
remote accesses 688
memory, viewing as typed object 137
mi interpreter 525
mi1 interpreter 525
mi2 interpreter 525
mi3 interpreter 525

MI commands in python 433
minimal language 249
minimal symbol dump 261
Minimal symbols and DLLs 327
MIPS addresses, masking 342
MIPS remote floating point 335
MIPS stack 340
miscellaneous settings 364
MMX registers (x86) 170
mode_t values, in file-i/o protocol 749
Modula-2 1
Modula-2 built-ins 233
Modula-2 checks 237
Modula-2 constants 234
Modula-2 defaults 237
Modula-2 operators 232
Modula-2 types 235
Modula-2, deviations from 237
Modula-2, GDB support 232
module functions and variables 260
modules 260
monitor commands, for `gdbserver` 307
Motorola 680x0 316
MS Windows debugging 326
MS-DOS system info 324
MS-DOS-specific commands 323
multiple locations, breakpoints 60
multiple processes 49
multiple targets 295
multiple threads 44
multiple threads, backtrace 109
multiple-symbols menu 138
multiprocess extensions, in remote protocol 715

N

name a thread 48
names of symbols 251
namespace in C++ 224
native Cygwin debugging 326
native DJGPP debugging 323
native script auto-loading 374
native target 297
negative breakpoint numbers 62
never read symbols 278
`New` *systag* message 44
new user interface 525
Newlib OS ABI and its influence on the longjmp handling 352
Nios II architecture 342
no debug info functions 270
no debug info variables 140
non-member C++ functions, set breakpoint in 58
non-stop mode 91
non-stop mode, and process record and replay 100
non-stop mode, and '`set displaced-stepping`' 673
non-stop mode, remote request 705
noninvasive task options 330
notational conventions, for GDB/MI 537
notification packets 735
notify output in GDB/MI 543
NULL elements in arrays 156
number of array elements to print 151
number representation 351
numbers for breakpoints 55

O

object files, relocatable, reading symbols from 279
Objective-C 228
Objective-C, classes and selectors 260
Objective-C, print objects 229
`objfile``-gdb.gdb` 521
`objfile``-gdb.py` 521
`objfile``-gdb.scm` 521
objfiles in guile 498
objfiles in python 441
observer debugging info 362
octal escapes in strings 156
online documentation 28
opaque data types 260
open flags, in file-i/o protocol 749
open, file-i/o system call 741
OpenCL C 229
OpenCL C Datatypes 229
OpenCL C Expressions 230
OpenCL C Operators 230
OpenRISC 1000 335
operate-and-get-next 346
operating system information 783
operating system information, process list 783
optimized code, debugging 185
optimized code, wrong values of variables 140
optimized out value in guile 476
optimized out value in Python 386
optimized out, in backtrace 110
optional debugging messages 359
optional warnings 357
OS ABI 352
OS information 171
out-of-line single-stepping 672
outermost frame 107
output formats 142
output syntax of GDB/MI 541
overlay area 211
overlay example program 215
overlays 211
overlays, setting breakpoints in 214
overloaded functions, calling 224
overloaded functions, overload resolution 226
overloading in C++ 225
overloading, Ada 242

P

'p' packet 688
'P' packet 688
packet acknowledgment, for GDB remote 737
packet size, remote protocol 713
packets, notification 735
packets, reporting on stdout 362
packets, tracepoint 725
page size 348
page tables display (MS-DOS) 324
pagination 348
parameters in guile 495
parameters in python 435
partial symbol dump 261
partial symbol tables, listing GDB's internal 261
Pascal 1
Pascal objects, static members display 158
Pascal support in GDB, limitations 231
pass signals to inferior, remote request 706
patching binaries 271
patching object files 277
pause current task (GNU Hurd) 330
pause current thread (GNU Hurd) 330
pauses in output 348
pending breakpoints 60
physical address from linear address 325
physname 359
pipe, `target remote` to 303
pipes 32
pointer values, in file-i/o protocol 747
pointer, finding referent 150
port rights, GNU Hurd 330
port sets, GNU Hurd 330
PowerPC architecture 342
prefix for data files 294
prefix for executable and shared library file names 283
premature return from system calls 93
preprocessor macro expansion, showing the results of 189
pretty print arrays 150
pretty print C++ virtual function tables 158
pretty-printer commands 160
print all frame argument values 151
print an Objective-C object description 229
print array indexes 150
print frame argument values for scalars only 151
print list of auto-loaded canned sequences of commands scripts 374
print list of auto-loaded Guile scripts 519
print list of auto-loaded Python scripts 465
print messages on inferior start and exit 43
print messages on thread start and exit 48
print messages when symbols are loaded 261
print settings 149
print structures in indented form 156
print/don't print memory addresses 149
printing byte arrays 143
printing data 133
printing frame argument values 151
printing frame information 154
printing memory tag violation information 155
printing nested structures 155
printing strings 143
probe static tracepoint marker 195
probing markers, static tracepoints 193
process detailed status information 323
process ID 322
process info via `/proc` 321
process list, QNX Neutrino 323
process record and replay 99
process status register 169
processes, multiple 49
`procfs` API calls 323
profiling GDB 679
program counter register 169
program entry point 110
programming in guile 469
programming in python 379
progspaces in guile 497
progspaces in python 439
prompt 345
protocol basics, file-i/o 739
protocol, GDB remote serial 683
protocol-specific representation of datatypes, in file-i/o protocol 747
python api 379
Python architectures 460
Python auto-loading 465
python commands 377
python commands, CLI 429
python commands, GDB/MI 433
python convenience functions 438
python directory 377
python exceptions 384
python finish breakpoints 458
python functions 379
python module 379
python modules 466
python pagination 379
python parameters 435
python pretty printing api 397
python scripting 377
python stdout 379
Python TUI Windows 463
Python, working with types 391
python, working with values from inferior 385

Q

'q' packet ... 689
'Q' packet ... 689
'QAllow' packet ... 699
'qAttached' packet ... 723
'qC' packet ... 699
'QCatchSyscalls' packet ... 706
'qCRC' packet ... 700
'QDisableRandomization' packet ... 700
'QEnvironmentHexEncoded' packet ... 701
'QEnvironmentReset' packet ... 702
'QEnvironmentUnset' packet ... 701
'qfThreadInfo' packet ... 702
'qGetTIBAddr' packet ... 703
'qGetTLSAddr' packet ... 703
'QMemTags' packet ... 704
'qMemTags' packet ... 704
'QNonStop' packet ... 705
'qOffsets' packet ... 705
'qP' packet ... 705
'QPassSignals' packet ... 706
'QProgramSignals' packet ... 707
'qRcmd' packet ... 708
'qSearch memory' packet ... 708
'qSearch:memory' packet ... 708
'QSetWorkingDir' packet ... 702
'QStartNoAckMode' packet ... 708
'QStartupWithShell' packet ... 700
'qsThreadInfo' packet ... 702
'qSupported' packet ... 709
'qSymbol' packet ... 717
'qTBuffer' packet ... 731
'QTBuffer size' packet ... 732
'QTDisable' packet ... 728
'QTDisconnected' packet ... 729
'QTDP' packet ... 725
'QTDPsrc' packet ... 727
'QTDV' packet ... 727
'QTEnable' packet ... 728
'qTfP' packet ... 730
'QTFrame' packet ... 727
'qTfSTM' packet ... 731
'qTfV' packet ... 731
'QThreadEvents' packet ... 707
'qThreadExtraInfo' packet ... 718
'QTinit' packet ... 728
'qTMinFTPILen' packet ... 728
'QTNotes' packet ... 732
'qTP' packet ... 730
'QTro' packet ... 729
'QTSave' packet ... 731
'qTsP' packet ... 730
'qTsSTM' packet ... 731
'QTStart' packet ... 728
'qTStatus' packet ... 729
'qTSTMat' packet ... 731
'QTStop' packet ... 728
'qTsV' packet ... 731
'qTV' packet ... 730
qualified thread ID ... 45
query attached, remote request ... 723
quotes in commands ... 26
quoting Ada internal identifiers ... 241
quoting names ... 251
'qXfer' packet ... 719

R

'r' packet ... 689
'R' packet ... 689
range checking ... 220
range stepping ... 83
ranged breakpoint ... 336
ranges of breakpoints ... 55
Ravenscar Profile ... 247
Ravenscar thread ... 247
raw printing ... 143
read special object, remote request ... 719
read, file-i/o system call ... 743
read-only sections ... 281
reading symbols from relocatable object files ... 279
reading symbols immediately ... 278
readline ... 345
Readline application name ... 346
receive rights, GNU Hurd ... 330
recent tracepoint number ... 196
record aggregates (Ada) ... 239
record mode ... 99
record serial communications on file ... 310
recording a session script ... 654
recording inferior's execution and replaying it ... 99
recordings in python ... 426
redirection ... 38
reference card ... 659
reference declarations ... 224
register cache, flushing ... 676
register packet format, MIPS ... 725
registers ... 169
Registers In Python ... 461
regular expression ... 58
reloading the overlay table ... 213
relocatable object files, reading symbols from ... 279
remote async notification debugging info ... 362
remote connection commands ... 301
remote connection without stubs ... 304
remote debugging ... 299
remote debugging, connecting ... 299
remote debugging, detach and program exit ... 299
remote debugging, symbol files ... 300
remote debugging, types of connections ... 299
remote memory comparison ... 146
remote packets, enabling and disabling ... 312
remote programs, interrupting ... 303
remote protocol debugging ... 362
remote protocol, binary data ... 683

remote protocol, field separator 683
remote query requests 699
remote serial debugging summary 318
remote serial debugging, overview 316
remote serial protocol 683
remote serial stub 317
remote serial stub list 316
remote serial stub, initialization 317
remote serial stub, main routine 317
remote stub, example 315
remote stub, support routines 317
remote target 296
remote target, file transfer 304
remote target, limit break- and watchpoints ... 310
remote target, limit watchpoints length 311
remote timeout 310
remove actions from a tracepoint 198
remove duplicate history 347
rename, file-i/o system call 744
Renesas 316
repeated array elements 154
repeating command sequences 23
repeating commands 23
replay log events, remote reply 696
replay mode 99
reporting bugs in GDB 653
reprint the last value 134, 272
reset environment, remote request 702
resources used by commands 680
response time, MIPS debugging 340
restart 52
restore data from a file 175
restrictions on Go expressions 228
result records in GDB/MI 544
resume threads of multiple processes simultaneously 90
resuming execution 80
returning from a function 268
reverse execution 97
rewind program state 52
run to first instruction 33
run to main procedure 33
run until specified location 81
running 32
running programs backward 97

S

's' packet 689
'S' packet 689
S12Z support 344
save breakpoints to a file for future sessions 77
save command history 346
save GDB output to a file 20
save tracepoints for future sessions 208
scheduler locking mode 90
scope 238
screen size 348
scripting commands 371
scripting with guile 468
scripting with python 377
search for a thread 48
search path for `libthread_db` 48
searching memory 181
searching memory, in remote debugging 708
searching source files 124
section offsets, remote request 705
segment descriptor tables 324
select Ctrl-C, BREAK or BREAK-g 311
select trace snapshot 205
selected frame 107
selecting guile pretty-printers 488
selecting python pretty-printers 398
self tests 677
semaphores on static probe points 77
send command to remote monitor 304
send command to simulator 332
send interrupt-sequence on start 311
send rights, GNU Hurd 330
send the output of a gdb command to a shell command 20
sending files to remote systems 304
separate debug sections 290
separate debugging information files 286
sequence-id, for GDB remote 683
serial connections, debugging 362
serial line, `target remote` 301
serial protocol, GDB remote 683
server prefix 642
`server`, command prefix 346
set ABI for MIPS 341
set breakpoints in many functions 58
set breakpoints on all functions 58
set environment variable, remote request 701
set exec-file-mismatch 39
set fast tracepoint 195
set inferior controlling terminal 39
set static tracepoint 195
set tdesc filename 769
set tracepoint 194
set working directory, remote request 702
setting variables 265
setting watchpoints 63
settings 24
`sh-stub.c` 316
shared libraries 281
shared library events, remote reply 696
shell command, exit code 165
shell command, exit signal 165
shell escape 19
show all convenience functions 169
show all user variables and functions 162
show exec-file-mismatch 40
show inferior's working directory 38
show last commands 348
show tdesc filename 769

SH 316
signals 86
signals the inferior may see, remote request 707
SIGQUIT signal, dump core of GDB 673
SingleKey keymap name 529
size of remote memory accesses 688
size of screen 348
skipping over files via glob-style patterns 84
skipping over functions and files 83
skipping over functions via
regular expressions 84
snapshot of a process 52
software watchpoints 63
source code, caching 676
source code, disable access 132
source file and line of a symbol 149
source line and its code address 128
source location 120
source path 124
Sparc 316
sparc-stub.c 316
Sparc64 support 343
sparcl-stub.c 316
SparcLite 316
Special Fortran commands 230
specifying location 120
SSE registers (x86) 170
stack frame 107
stack on Alpha 340
stack on MIPS 340
stack pointer register 169
stacking targets 295
standard registers 169
start a new independent interpreter 525
start a new trace experiment 202
starting 32
startup code, and backtrace 110
startup with shell, remote request 700
stat, file-i/o system call 745
static members of C++ objects 158
static members of Pascal objects 158
static probe point, DTrace 77
static probe point, SystemTap 77
static tracepoints 193
static tracepoints, in remote protocol 716
static tracepoints, setting 195
status of trace data collection 202
status output in GDB/MI 543
stepping 80
stepping and signal handlers 87
stepping into functions with no line info 81
stop a running trace experiment 202
stop on C++ exceptions 66
stop reply packets 695
stopped threads 89
store memory tags 704
stream records in GDB/MI 545
string tracing, in remote protocol 716
struct gdb_reader_funcs 647
struct gdb_symbol_callbacks 647
struct gdb_unwind_callbacks 647
struct return convention 339
struct stat, in file-i/o protocol 748
struct timeval, in file-i/o protocol 748
struct/union returned in registers 339
structure field name completion 26
stub example, remote debugging 315
stupid questions 358
styling 349
Super-H 337
supported GDB/MI features, list 630
supported packets, remote query 709
switching threads 44
switching threads automatically 90
symbol cache size 262
symbol cache, flushing 263
symbol cache, printing its contents 262
symbol cache, printing usage statistics 262
symbol decoding style, C++ 157
symbol dump 261
symbol file functions 363
symbol files, remote debugging 300
symbol from address 252
symbol lookup 363
symbol lookup, remote request 717
symbol names 251
symbol table 277
symbol table creation 363
symbol tables in guile 507
symbol tables in python 452
symbol tables, listing GDB's internal 261
symbol, source file and line 149
symbols in guile 503
symbols in python 448
symbols, never read 278
symbols, reading from
relocatable object files 279
symbols, reading immediately 278
Synopsys ARC 332
syscall DSO 280
system calls and thread breakpoints 93
system root, alternate 283
system, file-i/o system call 746
system-wide configuration scripts 669
system-wide init file 669

T

't' packet 689
'T' packet 689
'T' packet reply 695
tail call frames, debugging 186
target architecture 295
target byte order 298
target character set 177
target debugging info 363

target descriptions ... 769
target descriptions, AArch64 features ... 775
target descriptions, ARC Features ... 775
target descriptions, ARM features ... 776
target descriptions, enum types ... 774
target descriptions, i386 features ... 777
target descriptions, inclusion ... 770
target descriptions, LoongArch Features ... 778
target descriptions, M68K features ... 778
target descriptions, MicroBlaze features ... 778
target descriptions, MIPS features ... 778
target descriptions, NDS32 features ... 779
target descriptions, Nios II features ... 779
target descriptions, OpenRISC 1000 features ... 779
target descriptions, PowerPC features ... 779
target descriptions, predefined types ... 773
target descriptions, RISC-V Features ... 780
target descriptions, RX Features ... 781
target descriptions, S/390 features ... 781
target descriptions, sparc32 features ... 782
target descriptions, sparc64 features ... 782
target descriptions, standard features ... 775
target descriptions, System z features ... 781
target descriptions, TIC6x features ... 782
target descriptions, TMS320C6x features ... 782
target descriptions, XML format ... 769
target memory comparison ... 146
target output in GDB/MI ... 543
target stack description ... 677
target-assisted range stepping ... 83
task attributes (GNU Hurd) ... 330
task breakpoints, in Ada ... 246
task exception port, GNU Hurd ... 330
task suspend count ... 330
task switching with program using Ravenscar Profile ... 247
TCP port, `target remote` ... 302
temporarily change settings ... 24
terminal ... 38
Text User Interface ... 527
thread attributes info, remote request ... 718
thread breakpoints ... 93
thread breakpoints and system calls ... 93
thread create event, remote reply ... 697
thread create/exit events, remote request ... 707
thread default settings, GNU Hurd ... 331
thread exit event, remote reply ... 698
thread ID lists ... 45
thread identifier (GDB) ... 45
thread identifier (system) ... 44
thread info (Solaris) ... 46
thread information, remote request ... 705
thread list format ... 753
thread number, per inferior ... 45
thread properties, GNU Hurd ... 330
thread suspend count, GNU Hurd ... 331
thread-id, in remote protocol ... 684
threads and watchpoints ... 65
threads in python ... 424
threads of execution ... 44
threads, automatic switching ... 90
threads, continuing ... 89
threads, stopped ... 89
time of command execution ... 681
timeout for commands ... 682
timeout for serial communications ... 310
timeout, for remote target connection ... 312
timestamping debugging info ... 364
trace experiment, status of ... 202
trace file format ... 785
trace files ... 209
trace state variable value, remote request ... 730
trace state variables ... 197
traceback ... 108
traceframe info format ... 753
tracepoint actions ... 198
tracepoint conditions ... 197
tracepoint data, display ... 207
tracepoint deletion ... 196
tracepoint number ... 196
tracepoint packets ... 725
tracepoint pass count ... 196
tracepoint restrictions ... 204
tracepoint status, remote request ... 730
tracepoint variables ... 208
tracepoints ... 193
tracepoints support in `gdbserver` ... 308
trailing underscore, in Fortran symbols ... 230
translating between character sets ... 177
TUI ... 527
TUI commands ... 530
TUI configuration variables ... 532
TUI key bindings ... 528
TUI mouse support ... 530
TUI single key mode ... 529
type casting memory ... 137
type chain of a data type ... 677
type checking ... 219
type conversions in C++ ... 224
type printer ... 401
type printing API for Python ... 401
types in guile ... 482
types in Python ... 391

U

UDP port, `target remote` ... 303
union field name completion ... 26
unions in structures, printing ... 157
Unix domain socket ... 301
unknown address, locating ... 142
unknown type ... 256
unlink, file-i/o system call ... 745
unlinked object files ... 277
unload symbols from shared libraries ... 282
unmap an overlay ... 213

unmapped overlays 211
unset environment variable, remote request 701
unset tdesc filename 769
unsupported languages 249
unwind stack in called functions 269
unwind stack in called functions with unhandled exceptions 269
unwinding frames in Python 411
use only software watchpoints 64
user registers 676
user-defined command 367
user-defined macros 189
user-defined variables 162

V

value history 161
values from inferior, in guile 475
values from inferior, with Python 385
variable name conflict 139
variable object debugging info 364
variable objects in GDB/MI 587
variable values, wrong 140
variables, setting 265
'`vAttach`' packet 689
'`vCont`' packet 690
'`vCont?`' packet 691
'`vCtrlC`' packet 691
vector unit 171
vector, auxiliary 171
verbose operation 357
verify remote memory image 146
verify target memory image 146
'`vFile`' packet 691
'`vFlashDone`' packet 692
'`vFlashErase`' packet 691
'`vFlashWrite`' packet 691
vfork events, remote reply 697
vforkdone events, remote reply 697
virtual functions (C++) display 158
'`vKill`' packet 692
'`vMustReplyEmpty`' packet 692
volatile registers 170
'`vRun`' packet 692
'`vStopped`' packet 693
VTBL display 158

W

watchdog timer 682
watchpoints 55
watchpoints and threads 65
where to look for shared libraries 283
wild pointer, interpreting 150
Wind River Linux system-wide configuration script 670
word completion 24
working directory 124
working directory (of your program) 37
working language 217
write data into object, remote request 722
write, file-i/o system call 743
writing a frame filter 407
writing a Guile pretty-printer 489
writing a pretty-printer 399
writing convenience functions 438
writing into corefiles 271
writing into executables 271
writing JIT debug info readers 647
writing xmethods in Python 416
wrong values 140

X

`x` command, default address 128
'`X`' packet 693
Xilinx MicroBlaze 334
XInclude 770
XMD, Xilinx Microprocessor Debugger 334
xmethod API 414
xmethods in Python 414
XML parser debugging 364

Z

'`z`' packet 693
'`z0`' packet 693
'`z1`' packet 694
'`z2`' packet 694
'`z3`' packet 694
'`z4`' packet 695
'`Z`' packets 693
'`Z0`' packet 693
'`Z1`' packet 694
'`Z2`' packet 694
'`Z3`' packet 694
'`Z4`' packet 695

Command, Variable, and Function Index

!

! 19

#

(a comment) 23

$

$_, convenience variable 163
$__, convenience variable 163
$_ada_exception, convenience variable 67
$_any_caller_is, convenience function 168
$_any_caller_matches, convenience function 168
$_as_string, convenience function 168
$_caller_is, convenience function 167
$_caller_matches, convenience function 168
$_cimag, convenience function 168
$_creal, convenience function 168
$_exception, convenience variable 66
$_exitcode, convenience variable 163
$_exitsignal, convenience variable 163
$_gdb_maint_setting, convenience function ... 167
$_gdb_maint_setting_str, convenience function 167
$_gdb_major, convenience variable 165
$_gdb_minor, convenience variable 165
$_gdb_setting, convenience function 166
$_gdb_setting_str, convenience function 166
$_gthread, convenience variable 45
$_inferior, convenience variable 42
$_isvoid, convenience function 165
$_memeq, convenience function 167
$_probe_arg, convenience variable 78
$_regex, convenience function 167
$_sdata, collect 199
$_sdata, inspect, convenience variable 164
$_shell_exitcode, convenience variable 165
$_shell_exitsignal, convenience variable 165
$_siginfo, convenience variable 164
$_streq, convenience function 167
$_strlen, convenience function 167
$_thread, convenience variable 45
$_tlb, convenience variable 164
$bpnum, convenience variable 56
$cdir, convenience variable 124
$cwd, convenience variable 124
$tpnum 196
$trace_file 208
$trace_frame 208
$trace_func 208
$trace_line 208
$tracepoint 208

—

-ada-task-info 573
-add-inferior 636
-break-after 554
-break-commands 555
-break-condition 556
-break-delete 556
-break-disable 557
-break-enable 557
-break-info 558
-break-insert 558
-break-list 561
-break-passcount 562
-break-watch 562
-catch-assert 565
-catch-catch 569
-catch-exception 566
-catch-handlers 567
-catch-load 565
-catch-rethrow 568
-catch-throw 567
-catch-unload 565
-complete 638
-data-disassemble 598
-data-evaluate-expression 600
-data-list-changed-registers 601
-data-list-register-names 601
-data-list-register-values 602
-data-read-memory 603
-data-read-memory-bytes 605
-data-write-memory-bytes 606
-dprintf-insert 560
-enable-frame-filters 581
-enable-pretty-printing 589
-enable-timings 638
-environment-cd 570
-environment-directory 570
-environment-path 570
-environment-pwd 571
-exec-arguments 569
-exec-continue 574
-exec-finish 575
-exec-interrupt 576
-exec-jump 576
-exec-next 577
-exec-next-instruction 577
-exec-return 578
-exec-run 578
-exec-step 579
-exec-step-instruction 580
-exec-until 581
-file-exec-and-symbols 620
-file-exec-file 621
-file-list-exec-source-file 621

A

actions ... 198
ada-task-info ... 631
add-auto-load-safe-path ... 356
add-auto-load-scripts-directory ... 522
add-inferior ... 42
add-symbol-file ... 279
add-symbol-file-from-memory ... 280
adi assign ... 343
adi examine ... 343
advance *location* ... 82
alias ... 375
append ... 176
append-pretty-printer! ... 520
apropos ... 29
arch-bool-type ... 514
arch-char-type ... 513
arch-charset ... 513
arch-disassemble ... 515
arch-double-type ... 514
arch-float-type ... 514
arch-int-type ... 513
arch-int16-type ... 514
arch-int32-type ... 514
arch-int64-type ... 514
arch-int8-type ... 514
arch-long-type ... 513
arch-longdouble-type ... 514
arch-longlong-type ... 514
arch-name ... 513
arch-schar-type ... 513
arch-short-type ... 513
arch-uchar-type ... 513
arch-uint-type ... 513
arch-uint16-type ... 514
arch-uint32-type ... 514
arch-uint64-type ... 514
arch-uint8-type ... 514
arch-ulong-type ... 513
arch-ulonglong-type ... 514
arch-ushort-type ... 513
arch-void-type ... 513
arch-wide-charset ... 513
arch? ... 513
Architecture.disassemble ... 460
Architecture.integer_type ... 460
Architecture.name ... 460
Architecture.register_groups ... 461
Architecture.registers ... 461
attach ... 39
attach& ... 92
awatch ... 64

B

b (break) ... 56
backtrace ... 108
bfd caching ... 286
block-end ... 503
block-function ... 503
block-global-block ... 503
block-global? ... 503
block-start ... 502
block-static-block ... 503
block-static? ... 503
block-superblock ... 503
block-symbols ... 503
block-symbols-progress? ... 503
block-valid? ... 502
Block.end ... 448
Block.function ... 448
Block.global_block ... 448
Block.is_global ... 448
Block.is_static ... 448
Block.is_valid ... 448
Block.start ... 448
Block.static_block ... 448
Block.superblock ... 448
block? ... 502
BP_ACCESS_WATCHPOINT ... 456, 509
BP_BREAKPOINT ... 456, 509
BP_CATCHPOINT ... 456, 509
BP_HARDWARE_BREAKPOINT ... 456
BP_HARDWARE_WATCHPOINT ... 456, 509
BP_READ_WATCHPOINT ... 456, 509
BP_WATCHPOINT ... 456, 509
break ... 56
break ... task *taskno* (Ada) ... 246
break ... thread *thread-id* ... 93
break, and Objective-C ... 228
break-range ... 336
breakpoint annotation ... 644
breakpoint-commands ... 512
breakpoint-condition ... 511
breakpoint-enabled? ... 510
breakpoint-expression ... 510
breakpoint-hit-count ... 511
breakpoint-ignore-count ... 511
breakpoint-location ... 510
breakpoint-notifications ... 631
breakpoint-number ... 510
breakpoint-silent? ... 510
breakpoint-stop ... 511
breakpoint-task ... 511
breakpoint-temporary? ... 510
breakpoint-thread ... 511
breakpoint-type ... 510
breakpoint-valid? ... 510
breakpoint-visible? ... 510
Breakpoint.__init__ ... 455
Breakpoint.commands ... 458
Breakpoint.condition ... 458

Breakpoint.delete 457
Breakpoint.enabled 457
Breakpoint.expression 458
Breakpoint.hit_count 458
Breakpoint.ignore_count 457
Breakpoint.is_valid 457
Breakpoint.location 458
Breakpoint.number 457
Breakpoint.pending 457
Breakpoint.silent 457
Breakpoint.stop 456
Breakpoint.task 457
Breakpoint.temporary 458
Breakpoint.thread 457
Breakpoint.type 458
Breakpoint.visible 458
breakpoint? 509
BreakpointEvent.breakpoint 422
BreakpointEvent.breakpoints 422
breakpoints 509
breakpoints-invalid annotation 643
bt (backtrace) 108

C

c (continue) 80
c (SingleKey TUI key) 529
C-L 529
C-x 1 528
C-x 2 528
C-x a 528
C-x A 528
C-x C-a 528
C-x o 528
C-x s 528
call 269
catch 66
catch assert 67
catch catch 66
catch exception 67
catch exception unhandled 67
catch exec 67
catch fork 70
catch handlers 67
catch load 70
catch rethrow 66
catch signal 70
catch syscall 67
catch throw 66
catch unload 70
catch vfork 70
cd 38
cdir 124
checkpoint 52
clear 71
clear, and Objective-C 228
ClearObjFilesEvent.progspace 422
clone-inferior 42
collect (tracepoints) 199
Command.__init__ 430
Command.complete 430
Command.dont_repeat 430
Command.invoke 430
command? 491
COMMAND_BREAKPOINTS 432, 493
COMMAND_DATA 431, 493
COMMAND_FILES 431, 493
COMMAND_MAINTENANCE 432, 494
COMMAND_NONE 431, 493
COMMAND_OBSCURE 432, 494
COMMAND_RUNNING 431, 493
COMMAND_STACK 431, 493
COMMAND_STATUS 431, 493
COMMAND_SUPPORT 431, 493
COMMAND_TRACEPOINTS 432, 493
COMMAND_TUI 432
COMMAND_USER 432, 494
commands 74
commands annotation 642
compare-sections 146
compile code 271
compile file 272
complete 29
COMPLETE_COMMAND 432, 494
COMPLETE_EXPRESSION 433, 494
COMPLETE_FILENAME 432, 494
COMPLETE_LOCATION 432, 494
COMPLETE_NONE 432, 494
COMPLETE_SYMBOL 432, 494
condition 73
ConnectionEvent.connection 424
continue 80
continue& 92
core-file 278
ctf 209
Ctrl-o (operate-and-get-next) 23
current-arch 513
current-objfile 499
current-progspace 497
cwd 124

D

d (delete) 71
d (SingleKey TUI key) 529
data-directory 472
data-disassemble-a-option 631
data-read-memory-bytes 631
default-visualizer 488
define 368
define-prefix 368
delete 71
delete checkpoint *checkpoint-id* 52
delete display 148
delete mem 174
delete tracepoint 196

delete tvariable ... 198
delete-breakpoint! ... 509
demangle ... 253
detach ... 40
detach (remote) ... 303
detach inferiors *infno* ... 43
dir ... 125
directory ... 125
dis (disable) ... 72
disable ... 72
disable display ... 148
disable frame-filter ... 116
disable mem ... 174
disable pretty-printer ... 160
disable probes ... 78
disable tracepoint ... 196
disable type-printer ... 257
disassemble ... 128
disconnect ... 303
display ... 147
do (down) ... 112
document ... 368
dont-repeat ... 369, 492
Down ... 529
down ... 112
down-silently ... 113
dprintf ... 76
dprintf-style agent ... 76
dprintf-style call ... 76
dprintf-style gdb ... 76
dump ... 175

E

e (edit) ... 123
echo ... 372
edit ... 123
else ... 372
enable ... 72
enable display ... 148
enable frame-filter ... 117
enable mem ... 174
enable pretty-printer ... 160
enable probes ... 78
enable tracepoint ... 196
enable type-printer ... 257
enabled ... 415
enabled of type_printer ... 401
end (breakpoint commands) ... 74
end (if/else/while commands) ... 372
end (user-defined commands) ... 368
end-of-iteration ... 518
end-of-iteration? ... 518
error annotation ... 643
error-begin annotation ... 643
error-port ... 516
eval ... 374
EventRegistry.connect ... 420
EventRegistry.disconnect ... 420
exception-args ... 475
exception-key ... 475
exception? ... 475
exceptionHandler ... 318
exec-file ... 277
exec-file-mismatch ... 39
exec-run-start-option ... 631
execute ... 471
exit [*expression*] ... 19
exited annotation ... 643
ExitedEvent.exit_code ... 421
ExitedEvent.inferior ... 421
explore ... 135

F

f (frame) ... 111
f (SingleKey TUI key) ... 529
faas ... 116
fg (resume foreground execution) ... 80
field-artificial? ... 486
field-base-class? ... 486
field-bitpos ... 486
field-bitsize ... 486
field-enumval ... 486
field-name ... 485
field-type ... 486
field? ... 485
file ... 277
fin (finish) ... 81
find ... 181
find-pc-line ... 508
finish ... 81
finish& ... 92
FinishBreakpoint.__init__ ... 458
FinishBreakpoint.out_of_scope ... 459
FinishBreakpoint.return_value ... 459
flash-erase ... 297
flush_i_cache ... 318
flushregs ... 676
fo (forward-search) ... 124
focus ... 531
forward-search ... 124
frame address ... 111
frame apply ... 114
frame function ... 112
frame level ... 111
frame view ... 112
frame, selecting ... 111
frame-arch ... 499
frame-block ... 501
frame-function ... 501
frame-name ... 499
frame-newer ... 501
frame-older ... 501
frame-pc ... 501
frame-read-register ... 501

frame-read-var 501
frame-sal 501
frame-select 501
frame-type 499
frame-unwind-stop-reason 500
frame-valid? 499
Frame.architecture 444
Frame.block 445
Frame.find_sal 446
Frame.function 446
Frame.is_valid 444
Frame.level 446
Frame.name 444
Frame.newer 446
Frame.older 446
Frame.pc 445
Frame.read_register 446
Frame.read_var 446
Frame.select 446
Frame.type 444
Frame.unwind_stop_reason 445
frame? 499
FrameDecorator.address 405
FrameDecorator.elided 405
FrameDecorator.filename 405
FrameDecorator.frame_args 405
FrameDecorator.frame_locals 406
FrameDecorator.function 405
FrameDecorator.inferior_frame 407
FrameDecorator.line 405
FrameFilter.enabled 404
FrameFilter.filter 403
FrameFilter.name 404
FrameFilter.priority 404
frames-invalid annotation 643
frozen-varobjs 631
ftrace 195
Function 438
Function.__init__ 438
Function.invoke 438

G

gcore 176
gdb-object-kind 472
gdb-version 472
gdb.add_history 381
gdb.architecture_names 384
gdb.Block 447
gdb.block_for_pc 447
gdb.BP_ACCESS_WATCHPOINT 456
gdb.BP_BREAKPOINT 456
gdb.BP_CATCHPOINT 456
gdb.BP_HARDWARE_BREAKPOINT 456
gdb.BP_HARDWARE_WATCHPOINT 456
gdb.BP_READ_WATCHPOINT 456
gdb.BP_WATCHPOINT 456
gdb.Breakpoint 455
gdb.breakpoints 380
gdb.COMMAND_BREAKPOINTS 432
gdb.COMMAND_DATA 431
gdb.COMMAND_FILES 431
gdb.COMMAND_MAINTENANCE 432
gdb.COMMAND_NONE 431
gdb.COMMAND_OBSCURE 432
gdb.COMMAND_RUNNING 431
gdb.COMMAND_STACK 431
gdb.COMMAND_STATUS 431
gdb.COMMAND_SUPPORT 431
gdb.COMMAND_TRACEPOINTS 432
gdb.COMMAND_TUI 432
gdb.COMMAND_USER 432
gdb.COMPLETE_COMMAND 432
gdb.COMPLETE_EXPRESSION 433
gdb.COMPLETE_FILENAME 432
gdb.COMPLETE_LOCATION 432
gdb.COMPLETE_NONE 432
gdb.COMPLETE_SYMBOL 433
gdb.connections 384
gdb.convenience_variable 381
gdb.current_objfile 441
gdb.current_progspace 439
gdb.current_recording 426
gdb.decode_line 383
gdb.default_visualizer 398
gdb.error 384
gdb.execute 379
gdb.find_pc_line 381
gdb.FinishBreakpoint 458
gdb.flush 382
gdb.frame_stop_reason_string 444
gdb.FrameDecorator 404
gdb.Function 438
gdb.GdbError 384
gdb.history 380
gdb.history_count 381
gdb.host_charset 383
gdb.Inferior 419
gdb.InferiorCallPostEvent 423
gdb.InferiorCallPreEvent 422
gdb.inferiors 419
gdb.InferiorThread 424
gdb.invalidate_cached_frames 444
gdb.LazyString 459
gdb.LineTable 454
gdb.lookup_global_symbol 449
gdb.lookup_objfile 441
gdb.lookup_static_symbol 449
gdb.lookup_static_symbols 449
gdb.lookup_symbol 448, 449
gdb.lookup_type 391
gdb.MemoryError 384
gdb.newest_frame 443
gdb.Objfile 441
gdb.objfiles 441
gdb.PARAM_AUTO_BOOLEAN 437

gdb.PARAM_BOOLEAN 437
gdb.PARAM_ENUM 438
gdb.PARAM_FILENAME 438
gdb.PARAM_INTEGER 437
gdb.PARAM_OPTIONAL_FILENAME 437
gdb.PARAM_STRING 437
gdb.PARAM_STRING_NOESCAPE 437
gdb.PARAM_UINTEGER 437
gdb.PARAM_ZINTEGER 438
gdb.PARAM_ZUINTEGER 438
gdb.PARAM_ZUINTEGER_UNLIMITED 438
gdb.parameter 380
gdb.Parameter 435
gdb.parse_and_eval 381
gdb.post_event 381
gdb.pretty_printers 398
gdb.Progspace 439
gdb.progspaces 439
gdb.prompt_hook 383
gdb.PYTHONDIR 379
gdb.rbreak 380
gdb.register_window_type 463
gdb.search_memory 420
gdb.selected_frame 443
gdb.selected_inferior 419
gdb.selected_thread 424, 425
gdb.set_convenience_variable 381
gdb.set_parameter 380
gdb.solib_name 383
gdb.start_recording 426
gdb.STDERR 382
gdb.STDLOG 382, 383
gdb.STDOUT 382
gdb.stop_recording 426
gdb.string_to_argv 430
gdb.Symbol 448
gdb.SYMBOL_COMMON_BLOCK_DOMAIN 451
gdb.SYMBOL_LABEL_DOMAIN 451
gdb.SYMBOL_LOC_ARG 452
gdb.SYMBOL_LOC_BLOCK 452
gdb.SYMBOL_LOC_COMMON_BLOCK 452
gdb.SYMBOL_LOC_COMPUTED 452
gdb.SYMBOL_LOC_CONST 451
gdb.SYMBOL_LOC_CONST_BYTES 452
gdb.SYMBOL_LOC_LABEL 452
gdb.SYMBOL_LOC_LOCAL 452
gdb.SYMBOL_LOC_OPTIMIZED_OUT 452
gdb.SYMBOL_LOC_REF_ARG 452
gdb.SYMBOL_LOC_REGISTER 452
gdb.SYMBOL_LOC_REGPARM_ADDR 452
gdb.SYMBOL_LOC_STATIC 452
gdb.SYMBOL_LOC_TYPEDEF 452
gdb.SYMBOL_LOC_UNDEF 451
gdb.SYMBOL_LOC_UNRESOLVED 452
gdb.SYMBOL_MODULE_DOMAIN 451
gdb.SYMBOL_STRUCT_DOMAIN 451
gdb.SYMBOL_UNDEF_DOMAIN 451
gdb.SYMBOL_VAR_DOMAIN 451
gdb.Symtab 452
gdb.Symtab_and_line 452
gdb.target_charset 383
gdb.target_wide_charset 383
gdb.Type 391
gdb.TYPE_CODE_ARRAY 395
gdb.TYPE_CODE_BITSTRING 396
gdb.TYPE_CODE_BOOL 396
gdb.TYPE_CODE_CHAR 396
gdb.TYPE_CODE_COMPLEX 396
gdb.TYPE_CODE_DECFLOAT 396
gdb.TYPE_CODE_ENUM 395
gdb.TYPE_CODE_ERROR 396
gdb.TYPE_CODE_FLAGS 395
gdb.TYPE_CODE_FLT 396
gdb.TYPE_CODE_FUNC 395
gdb.TYPE_CODE_INT 396
gdb.TYPE_CODE_INTERNAL_FUNCTION 397
gdb.TYPE_CODE_MEMBERPTR 396
gdb.TYPE_CODE_METHOD 396
gdb.TYPE_CODE_METHODPTR 396
gdb.TYPE_CODE_NAMESPACE 396
gdb.TYPE_CODE_PTR 395
gdb.TYPE_CODE_RANGE 396
gdb.TYPE_CODE_REF 396
gdb.TYPE_CODE_RVALUE_REF 396
gdb.TYPE_CODE_SET 396
gdb.TYPE_CODE_STRING 396
gdb.TYPE_CODE_STRUCT 395
gdb.TYPE_CODE_TYPEDEF 396
gdb.TYPE_CODE_UNION 395
gdb.TYPE_CODE_VOID 396
gdb.unwinder.register_unwinder 413
gdb.UnwindInfo.add_saved_register 413
gdb.with_parameter 380
gdb.WP_ACCESS 456
gdb.WP_READ 456
gdb.WP_WRITE 456
gdb.write 382
gdb:error 474
gdb:invalid-object 474
gdb:memory-error 475
gdb:pp-type-error 475
gdb_init_reader 647
GdbExitingEvent.exit_code 424
gdbserver 304
generate-core-file 176
get-basic-type 520
getDebugChar 317
gnu_debuglink_crc32 289
gr 469
gu 469
guile 469
guile-data-directory 472
guile-repl 469

H

h (help) 28
handle 86
handle_exception 317
hbreak 57
help 28
help function 169
help target 296
help user-defined 369
history-append! 471
history-ref 471
hook 370
hookpost 370
host-config 472

I

i (info) 30
i (SingleKey TUI key) 529
if 372
ignore 74
inferior 41
inferior *infno* 42
Inferior.architecture 420
Inferior.connection 419
Inferior.connection_num 419
Inferior.is_valid 419
Inferior.num 419
Inferior.pid 419
Inferior.progspace 419
Inferior.read_memory 420
Inferior.search_memory 420
Inferior.thread_from_handle 420
Inferior.thread_from_thread_handle 420
Inferior.threads 419
Inferior.was_attached 419
Inferior.write_memory 420
InferiorCallPostEvent.address 423
InferiorCallPostEvent.ptid 423
InferiorCallPreEvent.address 423
InferiorCallPreEvent.ptid 423
InferiorThread.details 425
InferiorThread.global_num 425
InferiorThread.handle 426
InferiorThread.inferior 425
InferiorThread.is_exited 426
InferiorThread.is_running 426
InferiorThread.is_stopped 426
InferiorThread.is_valid 425
InferiorThread.name 425
InferiorThread.num 425
InferiorThread.ptid 425
InferiorThread.switch 426
info 30
info address 252
info all-registers 169
info args 113
info auto-load 353
info auto-load gdb-scripts 374
info auto-load guile-scripts 519
info auto-load libthread-db 355
info auto-load local-gdbinit 355
info auto-load python-scripts 465
info auxv 171
info breakpoints 59
info checkpoints 52
info classes 260
info common 230
info connections [*id*...] 41
info copying 30
info dcache 180
info display 148
info dll 282
info dos 324
info exceptions 242
info extensions 219
info f (info frame) 113
info files 280
info float 171
info frame 113
info frame, show the source language 219
info frame-filter 116
info functions 258
info handle 86
info inferiors [*id*...] 41
info io_registers, AVR 337
info line 128
info line, and Objective-C 228
info locals 114
info macro 189
info macros 189
info mem 174
info meminfo 323
info module 260
info modules 260
info os 172
info os cpus 172
info os files 172
info os modules 172
info os msg 172
info os processes 172
info os procgroups 172
info os semaphores 172
info os shm 173
info os sockets 173
info os threads 173
info pidlist 323
info pretty-printer 160
info probes 78
info proc 322
info program 55
info record 104
info registers 169
info scope 257
info selectors 260
info serial 325

info set ... 30
info share ... 282
info sharedlibrary ... 282
info signals ... 86
info skip ... 85
info source ... 258
info source, show the source language ... 219
info sources ... 258, 621
info stack ... 109
info static-tracepoint-markers ... 201
info symbol ... 252
info target ... 280
info task *taskno* ... 244
info tasks ... 243
info terminal ... 38
info threads ... 46
info tp [*n*...] ... 200
info tracepoints [*n*...] ... 200
info tvariables ... 198
info type-printers ... 257
info types ... 257
info variables ... 259
info vector ... 171
info w32 ... 326
info warranty ... 30
info watchpoints [*list*...] ... 64
info win ... 530
info-gdb-mi-command ... 631
init-if-undefined ... 162
input-port ... 515
inspect ... 133
instantiate on type_printer ... 401
Instruction.data ... 427
Instruction.decoded ... 427
Instruction.pc ... 427
Instruction.size ... 427
interpreter-exec ... 525
interrupt ... 93
iterator->list ... 518
iterator-filter ... 518
iterator-for-each ... 518
iterator-map ... 518
iterator-next! ... 518
iterator-object ... 518
iterator-progress ... 518
iterator-until ... 518
iterator? ... 518

J

j (jump) ... 266
jit-reader-load ... 647
jit-reader-unload ... 647
jump ... 266
jump, and Objective-C ... 228

K

KeyboardInterrupt ... 384
kill ... 40
kill inferiors *infno* ... 43
kvm ... 321

L

l (list) ... 119
language-option ... 631
layout ... 531
lazy-string->value ... 512
lazy-string-address ... 512
lazy-string-encoding ... 512
lazy-string-length ... 512
lazy-string-type ... 512
lazy-string? ... 512
LazyString.address ... 459
LazyString.encoding ... 459
LazyString.length ... 459
LazyString.type ... 460
LazyString.value ... 459
Left ... 529
LineTable.has_line ... 455
LineTable.line ... 454
LineTable.source_lines ... 455
LineTableEntry.line ... 454
LineTableEntry.pc ... 454
list ... 119
list, and Objective-C ... 228
load *filename offset* ... 297
lookup-block ... 503
lookup-global-symbol ... 505
lookup-symbol ... 505
lookup-type ... 482
loop_break ... 372
loop_continue ... 372

M

macro define ... 189
macro exp1 ... 189
macro expand ... 189
macro list ... 190
macro undef ... 190
maint ada set ignore-descriptive-types ... 249
maint ada show ignore-descriptive-types ... 249
maint agent ... 671
maint agent-eval ... 671
maint agent-printf ... 671
maint btrace clear ... 672
maint btrace clear-packet-history ... 672
maint btrace packet-history ... 671
maint check libthread-db ... 675
maint check xml-descriptions ... 675
maint check-psymtabs ... 673
maint check-symtabs ... 673
maint cplus first_component ... 673

maint cplus namespace 673
maint demangler-warning 673
maint deprecate 673
maint dump-me 673
maint expand-symtabs 673
maint flush dcache 181
maint flush register-cache 676
maint flush source-cache 676
maint flush symbol-cache 263
maint flush-symbol-cache 263
maint info bdccsr, S12Z 344
maint info bfds 286
maint info breakpoints 671
maint info btrace 671
maint info jit 672
maint info line-table 262
maint info program-spaces 43
maint info psymtabs 261
maint info sections 280
maint info selftests 677
maint info sol-threads 46
maint info symtabs 261
maint info target-sections 281
maint internal-error 673
maint internal-warning 673
maint packet 463, 674
maint print arc arc-instruction 332
maint print architecture 674
maint print c-tdesc 674
maint print cooked-registers 675
maint print core-file-backed-mappings 675
maint print dummy-frames 675
maint print msymbols 261
maint print objfiles 676
maint print psymbols 261
maint print raw-registers 675
maint print reggroups 676
maint print register-groups 675
maint print registers 675
maint print remote-registers 675
maint print section-scripts 677
maint print statistics 677
maint print symbol-cache 262
maint print symbol-cache-statistics 262
maint print symbols 261
maint print target-stack 677
maint print type 677
maint print unwind, HPPA 342
maint print user-registers 676
maint print xml-tdesc 675
maint selftest 677
maint set backtrace-on-fatal-signal 682
maint set bfd-sharing 286
maint set btrace pt skip-pad 672
maint set catch-demangler-crashes 673
maint set check-libthread-db 681
maint set demangler-warning 674
maint set dwarf always-disassemble 678
maint set dwarf max-cache-age 678
maint set dwarf unwinders 678
maint set gnu-source-highlight enabled 681
maint set internal-error 674
maint set internal-warning 674
maint set per-command 680
maint set profile 679
maint set selftest verbose 677
maint set show-all-tib 679
maint set show-debug-regs 679
maint set symbol-cache-size 262
maint set target-async 679
maint set target-non-stop *mode*
[on|off|auto] 679
maint set test-settings 682
maint set tui-resize-message 680
maint set worker-threads 678
maint show backtrace-on-fatal-signal 682
maint show bfd-sharing 286
maint show btrace pt skip-pad 672
maint show catch-demangler-crashes 673
maint show check-libthread-db 681
maint show demangler-warning 674
maint show dwarf always-disassemble 678
maint show dwarf max-cache-age 678
maint show dwarf unwinders 678
maint show gnu-source-highlight enabled 681
maint show internal-error 674
maint show internal-warning 674
maint show per-command 680
maint show profile 679
maint show selftest verbose 677
maint show show-all-tib 679
maint show show-debug-regs 679
maint show symbol-cache-size 262
maint show target-async 679
maint show target-non-stop 679
maint show
test-options-completion-result 682
maint show test-settings 682
maint show tui-resize-message 680
maint show worker-threads 678
maint space 681
maint test-options 681
maint time 681
maint translate-address 681
maint undeprecate 673
maint with 682
make 20
make-block-symbols-iterator 503
make-breakpoint 508
make-command 491
make-enum-hashtable 520
make-exception 475
make-field-iterator 483
make-iterator 517
make-lazy-value 480
make-list-iterator 518

make-parameter 495
make-pretty-printer 486
make-pretty-printer-worker 487
make-value 476
may-insert-breakpoints 95
may-insert-fast-tracepoints 95
may-insert-tracepoints 95
may-interrupt 95
may-write-memory 95
may-write-registers 94
mem 173
memory-port-range 516
memory-port-read-buffer-size 516
memory-port-write-buffer-size 516
memory-port? 516
memory-tag check 147
memory-tag print-allocation-tag 147
memory-tag print-logical-tag 147
memory-tag setatag 147
memory-tag with-logical-tag 147
MemoryChangedEvent.address 423
MemoryChangedEvent.length 423
memset 318
methods 415
MICommand.__init__ 433
MICommand.installed 434
MICommand.invoke 433
MICommand.name 434
monitor 304

N

n (next) 81
n (SingleKey TUI key) 529
name 414
name of type_printer 401
new-ui 525
newest-frame 501
NewInferiorEvent.inferior 424
NewObjFileEvent.new_objfile 422
NewThreadEvent.inferior_thread 424
next 81
next& 92
nexti 83
nexti& 92
ni (nexti) 83
nosharedlibrary 282

O

o (SingleKey TUI key) 529
Objfile 441
objfile-filename 498
objfile-pretty-printers 498
objfile-progspace 498
objfile-valid? 498
Objfile.add_separate_debug_file 443
Objfile.build_id 442
Objfile.filename 441
Objfile.frame_filters 442
Objfile.is_valid 443
Objfile.lookup_global_symbol 443
Objfile.lookup_static_symbol 443
Objfile.owner 442
Objfile.pretty_printers 442
Objfile.progspace 442
Objfile.type_printers 442
Objfile.username 442
objfile? 498
objfiles 499
observer 94
open-memory 516
output 373
output-port 516
overlay 213
overload-choice annotation 642

P

PARAM_AUTO_BOOLEAN 437, 496
PARAM_BOOLEAN 437, 496
PARAM_ENUM 438, 497
PARAM_FILENAME 438, 497
PARAM_INTEGER 437
PARAM_OPTIONAL_FILENAME 437, 497
PARAM_STRING 437, 497
PARAM_STRING_NOESCAPE 437, 497
PARAM_UINTEGER 437, 496
PARAM_ZINTEGER 438, 496
PARAM_ZUINTEGER 438, 497
PARAM_ZUINTEGER_UNLIMITED 438, 497
Parameter 435, 495
parameter-value 496
Parameter.__init__ 435
Parameter.get_set_string 436
Parameter.get_show_string 437
Parameter.set_doc 436
Parameter.show_doc 436
Parameter.value 436
parameter? 496
parse-and-eval 471
passcount 196
path 36
pending-breakpoints 631
PendingFrame.architecture 412
PendingFrame.create_unwind_info 412
PendingFrame.level 412

PendingFrame.read_register 411
PgDn 529
PgUp 529
pi 377
pipe 20
po (print-object) 229
post-commands annotation 642
post-overload-choice annotation 642
post-prompt annotation 642
post-prompt-for-continue annotation 642
post-query annotation 642
pre-commands annotation 642
pre-overload-choice annotation 642
pre-prompt annotation 642
pre-prompt-for-continue annotation 642
pre-query annotation 642
prepend-pretty-printer! 519
pretty-printer-enabled? 486
pretty-printer? 486
pretty-printers 486
pretty_printer.children 397
pretty_printer.display_hint 397
pretty_printer.to_string 398
print 133
print-object 229
printf 373
proc-trace-entry 323
proc-trace-exit 323
proc-untrace-entry 323
proc-untrace-exit 323
Progspace 439
progspace-filename 498
progspace-objfiles 498
progspace-pretty-printers 498
progspace-valid? 497
Progspace.block_for_pc 440
Progspace.filename 439
Progspace.find_pc_line 440
Progspace.frame_filters 439
Progspace.is_valid 440
Progspace.objfiles 440
Progspace.pretty_printers 439
Progspace.solib_name 440
Progspace.type_printers 439
progspace? 497
progspaces 497
prompt annotation 642
prompt-for-continue annotation 642
ptype 256
putDebugChar 317
pwd 38
py 377
python 377, 631
python-interactive 377

Q

q (quit) 19
q (SingleKey TUI key) 529
query annotation 642
queue-signal 267
quit [*expression*] 19
quit annotation 643

R

r (run) 32
r (SingleKey TUI key) 529
rbreak 58
rc (reverse-continue) 97
readnow 278
rec 99
rec btrace 99
rec btrace bts 99
rec btrace pt 99
rec bts 99
rec del 104
rec full 99
rec function-call-history 105
rec instruction-history 104
rec pt 99
rec s 100
recognize on type_recognizer 402
record 99
record btrace 99
record btrace bts 99
record btrace pt 99
record bts 99
record delete 104
record full 99
record function-call-history 105
record goto 100
record instruction-history 104
record pt 99
record restore 101
record save 101
record stop 100
Record.begin 426
Record.end 427
Record.format 426
Record.function_call_history 427
Record.goto 427
Record.instruction_history 427
Record.method 426
Record.replay_position 427
RecordFunctionSegment.instructions 428
RecordFunctionSegment.level 428
RecordFunctionSegment.next 428
RecordFunctionSegment.number 428
RecordFunctionSegment.prev 428
RecordFunctionSegment.symbol 428
RecordFunctionSegment.up 428
RecordGap.error_code 428
RecordGap.error_string 428

RecordGap.number ... 428
RecordInstruction.is_speculative ... 427
RecordInstruction.number ... 427
RecordInstruction.sal ... 427
refresh ... 531
register-breakpoint! ... 509
register-command! ... 491
register-parameter! ... 496
register_xmethod_matcher ... 416
RegisterChangedEvent.frame ... 423
RegisterChangedEvent.regnum ... 423
RegisterDescriptor.name ... 461
RegisterDescriptorIterator.find ... 461
RegisterGroup.name ... 461
remote delete ... 304
remote get ... 304
remote put ... 304
RemoteTargetConnection.send_packet ... 463
remove-inferiors ... 42
remove-symbol-file ... 279
restart *checkpoint-id* ... 52
restore ... 176
RET (repeat last command) ... 23
return ... 268
reverse-continue ... 97
reverse-finish ... 98
reverse-next ... 98
reverse-nexti ... 98
reverse-search ... 124
reverse-step ... 97
reverse-stepi ... 97
Right ... 529
rn (reverse-next) ... 98
rni (reverse-nexti) ... 98
rs (step) ... 97
rsi (reverse-stepi) ... 97
run ... 32
run& ... 92
rwatch ... 64

S

s (SingleKey TUI key) ... 529
s (step) ... 80
sal-last ... 508
sal-line ... 507
sal-pc ... 508
sal-symtab ... 507
sal-valid? ... 507
sal? ... 507
save breakpoints ... 77
save gdb-index ... 291
save tracepoints ... 208
save-tracepoints ... 208
search ... 124
section ... 280
select-frame ... 113
selected-frame ... 501
self ... 492
set ... 30
set ada print-signatures ... 242
set ada source-charset ... 247
set ada trust-PAD-over-XVS ... 248
set agent off ... 649
set agent on ... 649
set annotate ... 641
set architecture ... 295
set args ... 36
set arm ... 333
set auto-connect-native-target ... 34
set auto-load gdb-scripts ... 374
set auto-load guile-scripts ... 519
set auto-load libthread-db ... 355
set auto-load local-gdbinit ... 355
set auto-load off ... 353
set auto-load python-scripts ... 465
set auto-load safe-path ... 356
set auto-load scripts-directory ... 521
set auto-solib-add ... 282
set backtrace ... 110
set basenames-may-differ ... 285
set breakpoint always-inserted ... 62
set breakpoint auto-hw ... 61
set breakpoint condition-evaluation ... 62
set breakpoint pending ... 61
set build-id-verbose ... 287
set can-use-hw-watchpoints ... 64
set case-sensitive ... 251
set charset ... 178
set check range ... 220
set check type ... 220
set circular-trace-buffer ... 203
set code-cache ... 180
set coerce-float-to-double ... 352
set com1base ... 325
set com1irq ... 325
set com2base ... 325
set com2irq ... 325
set com3base ... 325
set com3irq ... 325
set com4base ... 325
set com4irq ... 325
set complaints ... 358
set confirm ... 358
set cp-abi ... 352
set cwd ... 38
set cygwin-exceptions ... 326
set data-directory ... 294
set dcache line-size ... 181
set dcache size ... 181
set debug ... 359
set debug aarch64 ... 338
set debug arc ... 332
set debug auto-load ... 357
set debug bfd-cache *level* ... 286
set debug darwin ... 331

set debug entry-values ... 186
set debug hppa ... 342
set debug libthread-db ... 49
set debug mach-o ... 331
set debug mips ... 342
set debug monitor ... 297
set debug nios2 ... 343
set debug py-breakpoint ... 378
set debug py-unwind ... 378
set debug skip ... 86
set debug threads ... 49
set debug-file-directory ... 287
set debugevents ... 327
set debugexceptions ... 327
set debugexec ... 327
set debuginfod enabled ... 791
set debuginfod urls ... 791
set debuginfod verbose ... 791
set debugmemory ... 327
set default-collect ... 200
set demangle-style ... 157
set detach-on-fork ... 50
set directories ... 127
set disable-randomization ... 35
set disassemble-next-line ... 132
set disassembler-options ... 131
set disassembly-flavor ... 131
set disconnected-dprintf ... 77
set disconnected-tracing ... 203
set displaced-stepping ... 672
set dump-excluded-mappings ... 177
set editing ... 345
set endian ... 298
set environment ... 37
set exceptions, Hurd command ... 329
set exec-direction ... 98
set exec-done-display ... 359
set exec-wrapper ... 33
set extended-prompt ... 345
set extension-language ... 219
set follow-exec-mode ... 51
set follow-fork-mode ... 50
set fortran repack-array-slices ... 230
set frame-filter priority ... 118
set gnutarget ... 296
set guile print-stack ... 473
set hash, for remote monitors ... 297
set height ... 348
set history expansion ... 347
set history filename ... 346
set history remove-duplicates ... 347
set history save ... 346
set history size ... 347
set host-charset ... 178
set index-cache ... 292
set inferior-tty ... 39
set input-radix ... 351
set interactive-mode ... 364
set language ... 218
set libthread-db-search-path ... 48
set listsize ... 119
set logging enabled ... 21
set mach-exceptions ... 332
set max-completions ... 25
set max-user-call-depth ... 369
set max-value-size ... 183
set may-call-functions ... 270
set mem inaccessible-by-default ... 175
set mi-async ... 539
set mips abi ... 341
set mips compression ... 341
set mips mask-address ... 342
set mipsfpu ... 335
set mpx bound ... 340
set multiple-symbols ... 138
set new-console ... 327
set new-group ... 327
set non-stop ... 91
set opaque-type-resolution ... 260
set osabi ... 352
set output-radix ... 351
set overload-resolution ... 226
set pagination ... 348
set powerpc ... 336
set print ... 149
set print entry-values ... 152
set print finish ... 81
set print frame-arguments ... 151
set print frame-info ... 154
set print inferior-events ... 43
set print symbol-loading ... 261
set print thread-events ... 48
set print type hex ... 252
set print type methods ... 251
set print type nested-type-limit ... 251
set print type typedefs ... 252
set processor ... 295
set procfs-file ... 323
set procfs-trace ... 323
set prompt ... 345
set python dont-write-bytecode ... 378
set python ignore-environment ... 378
set python print-stack ... 377
set radix ... 351
set range-stepping ... 83
set ravenscar task-switching off ... 247
set ravenscar task-switching on ... 247
set record ... 105
set record btrace ... 102
set record btrace bts ... 103
set record btrace pt ... 103
set record full ... 101
set remote ... 309
set remote system-call-allowed ... 747
set remote-mips64-transfers-32bit-regs ... 342
set remotecache ... 180

set remoteflow ... 310
set schedule-multiple ... 91
set script-extension ... 367
set sh calling-convention ... 337
set shell ... 327
set signal-thread ... 329
set signals, Hurd command ... 329
set sigs, Hurd command ... 329
set sigthread ... 329
set solib-absolute-prefix ... 283
set solib-search-path ... 284
set source open ... 132
set stack-cache ... 180
set startup-quietly ... 13
set startup-with-shell ... 34
set step-mode ... 81
set stop-on-solib-events ... 283
set stopped, Hurd command ... 329
set struct-convention ... 339
set style ... 349
set substitute-path ... 127
set suppress-cli-notifications ... 364
set sysroot ... 283
set target-charset ... 177
set target-file-system-kind (unix|dos-based|auto) ... 284
set target-wide-charset ... 178
set task, Hurd commands ... 330
set tcp ... 311
set thread, Hurd command ... 330
set trace-buffer-size ... 203
set trace-commands ... 358
set trace-notes ... 204
set trace-stop-notes ... 204
set trace-user ... 204
set trust-readonly-sections ... 281
set tui active-border-mode ... 532
set tui border-kind ... 532
set tui border-mode ... 532
set tui compact-source ... 533
set tui tab-width ... 533
set unwind-on-terminating-exception ... 269
set unwindonsignal ... 269
set use-coredump-filter ... 176
set variable ... 265
set verbose ... 358
set watchdog ... 682
set width ... 348
set write ... 271
set-breakpoint-condition! ... 511
set-breakpoint-enabled! ... 510
set-breakpoint-hit-count! ... 511
set-breakpoint-ignore-count! ... 511
set-breakpoint-silent! ... 510
set-breakpoint-stop! ... 511
set-breakpoint-task! ... 511
set-breakpoint-thread! ... 511
set-iterator-progress! ... 518
set-memory-port-read-buffer-size! ... 516
set-memory-port-write-buffer-size! ... 517
set-objfile-pretty-printers! ... 499
set-parameter-value! ... 496
set-pretty-printer-enabled! ... 486
set-pretty-printers! ... 486
set-progspace-pretty-printers! ... 498
set_debug_traps ... 317
share ... 282
sharedlibrary ... 282
shell ... 19
show ... 30
show ada print-signatures ... 242
show ada source-charset ... 247
show ada trust-PAD-over-XVS ... 248
show agent ... 649
show annotate ... 641
show architecture ... 295
show args ... 36
show arm ... 333
show auto-load ... 353
show auto-load gdb-scripts ... 374
show auto-load guile-scripts ... 519
show auto-load libthread-db ... 355
show auto-load local-gdbinit ... 355
show auto-load python-scripts ... 465
show auto-load safe-path ... 356
show auto-load scripts-directory ... 522
show auto-solib-add ... 282
show backtrace ... 110
show basenames-may-differ ... 286
show breakpoint always-inserted ... 62
show breakpoint auto-hw ... 61
show breakpoint condition-evaluation ... 62
show breakpoint pending ... 61
show build-id-verbose ... 287
show can-use-hw-watchpoints ... 64
show case-sensitive ... 251
show charset ... 178
show check range ... 220
show check type ... 220
show circular-trace-buffer ... 203
show code-cache ... 180
show coerce-float-to-double ... 352
show com1base ... 325
show com1irq ... 325
show com2base ... 325
show com2irq ... 325
show com3base ... 325
show com3irq ... 325
show com4base ... 325
show com4irq ... 325
show commands ... 348
show complaints ... 358
show configuration ... 30
show confirm ... 358
show convenience ... 162
show copying ... 30

show cp-abi ... 352
show cwd ... 38
show cygwin-exceptions ... 327
show data-directory ... 294
show dcache line-size ... 181
show dcache size ... 181
show debug ... 359
show debug arc ... 332
show debug auto-load ... 357
show debug bfd-cache ... 286
show debug darwin ... 331
show debug entry-values ... 187
show debug libthread-db ... 49
show debug mach-o ... 331
show debug mips ... 342
show debug monitor ... 297
show debug nios2 ... 343
show debug py-breakpoint ... 378
show debug py-unwind ... 378
show debug skip ... 86
show debug threads ... 49
show debug-file-directory ... 287
show debuginfod enabled ... 791
show debuginfod urls ... 791
show debuginfod verbose ... 791
show default-collect ... 200
show detach-on-fork ... 50
show directories ... 127
show disassemble-next-line ... 132
show disassembler-options ... 131
show disassembly-flavor ... 131
show disconnected-dprintf ... 77
show disconnected-tracing ... 203
show displaced-stepping ... 672
show editing ... 346
show environment ... 37
show exceptions, Hurd command ... 330
show exec-done-display ... 359
show extended-prompt ... 345
show follow-fork-mode ... 50
show fortran repack-array-slices ... 230
show frame-filter priority ... 118
show gnutarget ... 296
show hash, for remote monitors ... 297
show height ... 348
show history ... 347
show host-charset ... 178
show index-cache ... 292
show inferior-tty ... 39
show input-radix ... 351
show interactive-mode ... 364
show language ... 219
show libthread-db-search-path ... 49
show listsize ... 119
show logging ... 21
show mach-exceptions ... 332
show max-completions ... 25
show max-user-call-depth ... 369
show max-value-size ... 183
show may-call-functions ... 270
show mem inaccessible-by-default ... 175
show mi-async ... 539
show mips abi ... 341
show mips compression ... 341
show mips mask-address ... 342
show mipsfpu ... 335
show mpx bound ... 340
show multiple-symbols ... 138
show new-console ... 327
show new-group ... 327
show non-stop ... 91
show opaque-type-resolution ... 261
show osabi ... 352
show output-radix ... 351
show overload-resolution ... 226
show pagination ... 349
show paths ... 36
show print ... 149
show print finish ... 81
show print inferior-events ... 43
show print symbol-loading ... 261
show print thread-events ... 48
show print type hex ... 252
show print type methods ... 251
show print type nested-type-limit ... 251
show print type typedefs ... 252
show processor ... 295
show procfs-file ... 323
show procfs-trace ... 323
show prompt ... 345
show radix ... 351
show range-stepping ... 83
show ravenscar task-switching ... 247
show record ... 105
show record btrace ... 103
show record full ... 101
show remote ... 309
show remote system-call-allowed ... 747
show remote-mips64-transfers-32bit-regs ... 342
show remotecache ... 180
show remoteflow ... 310
show script-extension ... 367
show sh calling-convention ... 338
show shell ... 327
show signal-thread ... 329
show signals, Hurd command ... 329
show sigs, Hurd command ... 329
show sigthread ... 329
show solib-search-path ... 284
show source open ... 132
show stack-cache ... 180
show startup-quietly ... 13
show stop-on-solib-events ... 283
show stopped, Hurd command ... 329
show struct-convention ... 339
show style ... 349

show substitute-path ... 127
show suppress-cli-notifications ... 365
show sysroot ... 284
show target-charset ... 178
show target-file-system-kind ... 284
show target-wide-charset ... 178
show task, Hurd commands ... 330
show tcp ... 311
show thread, Hurd command ... 330
show trace-buffer-size ... 204
show trace-notes ... 204
show trace-stop-notes ... 204
show trace-user ... 204
show unwind-on-terminating-exception ... 270
show unwindonsignal ... 269
show user ... 369
show values ... 162
show verbose ... 358
show version ... 30
show warranty ... 30
show width ... 348
show write ... 271
si (stepi) ... 83
signal ... 267
signal annotation ... 644
signal-event ... 326
signal-name annotation ... 643
signal-name-end annotation ... 643
signal-string annotation ... 643
signal-string-end annotation ... 643
SignalEvent.stop_signal ... 422
signalled annotation ... 643
silent ... 75
sim, a command ... 332
skip ... 84
skip delete ... 85
skip disable ... 85
skip enable ... 85
skip file ... 85
skip function ... 85
source ... 371
source annotation ... 644
start ... 33
starti ... 33
starting annotation ... 643
STDERR ... 382
stdio-port? ... 516
STDLOG ... 382, 383
STDOUT ... 382
step ... 80
step& ... 92
stepi ... 83
stepi& ... 92
stop, a pseudo-command ... 370
stopping annotation ... 643
strace ... 195
string->argv ... 492
symbol-addr-class ... 504
symbol-argument? ... 504
symbol-constant? ... 504
symbol-file ... 277
symbol-function? ... 504
symbol-line ... 504
symbol-linkage-name ... 504
symbol-name ... 504
symbol-needs-frame? ... 504
symbol-print-name ... 504
symbol-symtab ... 504
symbol-type ... 504
symbol-valid? ... 504
symbol-value ... 505
symbol-variable? ... 505
Symbol.addr_class ... 450
Symbol.is_argument ... 450
Symbol.is_constant ... 450
Symbol.is_function ... 451
Symbol.is_valid ... 451
Symbol.is_variable ... 451
Symbol.line ... 450
Symbol.linkage_name ... 450
Symbol.name ... 450
Symbol.needs_frame ... 450
Symbol.print_name ... 450
Symbol.symtab ... 450
Symbol.type ... 450
Symbol.value ... 451
symbol? ... 504
SYMBOL_COMMON_BLOCK_DOMAIN ... 451
SYMBOL_FUNCTIONS_DOMAIN ... 506
SYMBOL_LABEL_DOMAIN ... 451, 505
SYMBOL_LOC_ARG ... 452, 506
SYMBOL_LOC_BLOCK ... 452, 506
SYMBOL_LOC_COMMON_BLOCK ... 452
SYMBOL_LOC_COMPUTED ... 452, 506
SYMBOL_LOC_CONST ... 451, 506
SYMBOL_LOC_CONST_BYTES ... 452, 506
SYMBOL_LOC_LABEL ... 452
SYMBOL_LOC_LOCAL ... 452, 506
SYMBOL_LOC_OPTIMIZED_OUT ... 452, 506
SYMBOL_LOC_REF_ARG ... 452, 506
SYMBOL_LOC_REGISTER ... 452, 506
SYMBOL_LOC_REGPARM_ADDR ... 452, 506
SYMBOL_LOC_STATIC ... 451, 506
SYMBOL_LOC_TYPEDEF ... 452, 506
SYMBOL_LOC_UNDEF ... 451, 506
SYMBOL_LOC_UNRESOLVED ... 452, 506
SYMBOL_MODULE_DOMAIN ... 451
SYMBOL_STRUCT_DOMAIN ... 451, 505
SYMBOL_TYPES_DOMAIN ... 506
SYMBOL_UNDEF_DOMAIN ... 451, 505
SYMBOL_VAR_DOMAIN ... 451, 505
SYMBOL_VARIABLES_DOMAIN ... 505
symtab-filename ... 507
symtab-fullname ... 507
symtab-global-block ... 507
symtab-objfile ... 507

symtab-static-block 507
symtab-valid? 507
Symtab.filename 453
Symtab.fullname 453
Symtab.global_block 453
Symtab.is_valid 453
Symtab.linetable 454
Symtab.objfile 453
Symtab.producer 453
Symtab.static_block 454
symtab? 507
Symtab_and_line.is_valid 453
Symtab_and_line.last 453
Symtab_and_line.line 453
Symtab_and_line.pc 453
Symtab_and_line.symtab 453
sysinfo 324

T

taas 47
tabset 533
target 296
target ctf 209
target record 99
target record-btrace 99
target record-full 99
target sim 335
target tfile 209
target-config 472
TargetConnection.description 462
TargetConnection.details 463
TargetConnection.is_valid 462
TargetConnection.num 462
TargetConnection.type 462
task (Ada) 244
tbreak 57
tcatch 70
tdump 207
teval (tracepoints) 200
tfaas 47
tfile 209
tfind 205
thbreak 58
this, inside C++ member functions 224
thread apply 47
thread find 48
thread name 48
thread *thread-id* 46
thread-info 631
ThreadEvent.inferior_thread 421
throw-user-error 492
trace 194
tsave 209
tstart [*notes*] 202
tstatus 202
tstop [*notes*] 202
tty 38
tui disable 530
tui enable 530
tui new-layout 530
tui reg 531
TuiWindow.erase 464
TuiWindow.height 464
TuiWindow.is_valid 464
TuiWindow.title 464
TuiWindow.width 464
TuiWindow.write 464
tvariable 198
type-array 482
type-code 482
type-const 483
type-field 484
type-fields 483
type-has-field-deep? 520
type-has-field? 484
type-name 482
type-num-fields 483
type-pointer 483
type-print-name 482
type-range 483
type-reference 483
type-sizeof 482
type-strip-typedefs 482
type-tag 482
type-target 483
type-unqualified 483
type-vector 482
type-volatile 483
Type.alignof 392
Type.array 394
Type.code 392
Type.const 394
Type.dynamic 392
Type.fields 393
Type.is_scalar 393
Type.is_signed 393
Type.name 392
Type.objfile 393
Type.optimized_out 395
Type.pointer 394
Type.range 394
Type.reference 394
Type.sizeof 392
Type.strip_typedefs 395
Type.tag 392
Type.target 395
Type.template_argument 395
Type.unqualified 394
Type.vector 394
Type.volatile 394
type? 482
TYPE_CODE_ARRAY 395, 484
TYPE_CODE_BITSTRING 396, 485
TYPE_CODE_BOOL 396, 485
TYPE_CODE_CHAR 396, 485

TYPE_CODE_COMPLEX 396, 485
TYPE_CODE_DECFLOAT 396, 485
TYPE_CODE_ENUM 395, 484
TYPE_CODE_ERROR 396, 485
TYPE_CODE_FLAGS 395, 484
TYPE_CODE_FLT 396, 484
TYPE_CODE_FUNC 395, 484
TYPE_CODE_INT 395, 484
TYPE_CODE_INTERNAL_FUNCTION 396, 485
TYPE_CODE_MEMBERPTR 396, 485
TYPE_CODE_METHOD 396, 485
TYPE_CODE_METHODPTR 396, 485
TYPE_CODE_NAMESPACE 396, 485
TYPE_CODE_PTR 395, 484
TYPE_CODE_RANGE 396, 484
TYPE_CODE_REF 396, 485
TYPE_CODE_RVALUE_REF 396, 485
TYPE_CODE_SET 396, 484
TYPE_CODE_STRING 396, 484
TYPE_CODE_STRUCT 395, 484
TYPE_CODE_TYPEDEF 396, 485
TYPE_CODE_UNION 395, 484
TYPE_CODE_VOID 396, 484

U

u (SingleKey TUI key) 529
u (until) 81
undefined-command-error-code 631
undisplay 148
unset environment 37
unset substitute-path 127
until 81
until& 92
unwind-stop-reason-string 501
up 112
Up 529
up-silently 113
update 532

V

v (SingleKey TUI key) 529
value->bool 479
value->bytevector 479
value->integer 479
value->lazy-string 480
value->real 479
value->string 479
value-abs 481
value-add 481
value-address 477
value-call 479
value-cast 477
value-const-value 478
value-dereference 477
value-div 481
value-dynamic-cast 477
value-dynamic-type 477
value-fetch-lazy! 480
value-field 479
value-lazy? 480
value-logand 481
value-logior 481
value-lognot 481
value-logxor 481
value-lsh 481
value-max 481
value-min 481
value-mod 481
value-mul 481
value-neg 481
value-not 481
value-optimized-out? 476
value-pos 481
value-pow 481
value-print 480
value-reference-value 478
value-referenced-value 478
value-reinterpret-cast 477
value-rem 481
value-rsh 481
value-rvalue-reference-value 478
value-sub 481
value-subscript 479
value-type 477
Value.__init__ 386, 387
Value.address 386
Value.cast 387
Value.const_value 388
Value.dereference 387
Value.dynamic_cast 388
Value.dynamic_type 386
Value.fetch_lazy 391
Value.format_string 389
Value.is_lazy 386
Value.is_optimized_out 386
Value.lazy_string 391
Value.reference_value 388
Value.referenced_value 388
Value.reinterpret_cast 389
Value.string 390
Value.type 386
value<=? 481
value<? 481
value=? 481
value>=? 481
value>? 481
value? 475

W

w (SingleKey TUI key) 529
w (with) 24
watch 63
watchpoint annotation 644
whatis 253
where 109
while 372
while-stepping (tracepoints) 200
Window.click 465
Window.close 464
Window.hscroll 465
Window.render 464
Window.vscroll 465
winheight 532
with command 24
WP_ACCESS 456, 509
WP_READ 456, 509
WP_WRITE 456, 509

X

x (examine memory) 143
x(examine), and info line 128
XMethod.__init__ 415
XMethodMatcher.__init__ 415
XMethodMatcher.match 415
XMethodWorker.__call__ 415
XMethodWorker.get_arg_types 415
XMethodWorker.get_result_type 415

The body of this manual is set in
cmr10,
with headings in **cmbx10**
and examples in `cmtt10`.
cmti10,
cmbx10, and
cmsl10
are used for emphasis.