GDB Internals

A guide to the internals of the GNU debugger

John Gilmore
Cygnus Solutions
Second Edition:
Stan Shebs
Cygnus Solutions

Cygnus Solutions
Revision: 1.127
TeXinfo 2.257

Scope of this Document

This document documents the internals of the GNU debugger, GDB. It includes description of GDB's key algorithms and operations, as well as the mechanisms that adapt GDB to specific hosts and targets.

1 Requirements

Before diving into the internals, you should understand the formal requirements and other expectations for GDB. Although some of these may seem obvious, there have been proposals for GDB that have run counter to these requirements.

First of all, GDB is a debugger. It's not designed to be a front panel for embedded systems. It's not a text editor. It's not a shell. It's not a programming environment.

GDB is an interactive tool. Although a batch mode is available, GDB's primary role is to interact with a human programmer.

GDB should be responsive to the user. A programmer hot on the trail of a nasty bug, and operating under a looming deadline, is going to be very impatient of everything, including the response time to debugger commands.

GDB should be relatively permissive, such as for expressions. While the compiler should be picky (or have the option to be made picky), since source code lives for a long time usually, the programmer doing debugging shouldn't be spending time figuring out to mollify the debugger.

GDB will be called upon to deal with really large programs. Executable sizes of 50 to 100 megabytes occur regularly, and we've heard reports of programs approaching 1 gigabyte in size.

GDB should be able to run everywhere. No other debugger is available for even half as many configurations as GDB supports.

2 Overall Structure

GDB consists of three major subsystems: user interface, symbol handling (the "symbol side"), and target system handling (the "target side").

Ther user interface consists of several actual interfaces, plus supporting code.

The symbol side consists of object file readers, debugging info interpreters, symbol table management, source language expression parsing, type and value printing.

The target side consists of execution control, stack frame analysis, and physical target manipulation.

The target side/symbol side division is not formal, and there are a number of exceptions. For instance, core file support involves symbolic elements (the basic core file reader is in BFD) and target elements (it supplies the contents of memory and the values of registers). Instead, this division is useful for understanding how the minor subsystems should fit together.

2.1 The Symbol Side

The symbolic side of GDB can be thought of as "everything you can do in GDB without having a live program running". For instance, you can look at the types of variables, and evaluate many kinds of expressions.

2.2 The Target Side

The target side of GDB is the "bits and bytes manipulator". Although it may make reference to symbolic info here and there, most of the target side will run with only a stripped executable available – or even no executable at all, in remote debugging cases.

Operations such as disassembly, stack frame crawls, and register display, are able to work with no symbolic info at all. In some cases, such as disassembly, GDB will use symbolic info to present addresses relative to symbols rather than as raw numbers, but it will work either way.

2.3 Configurations

Host refers to attributes of the system where GDB runs. *Target* refers to the system where the program being debugged executes. In most cases they are the same machine, in which case a third type of *Native* attributes come into play.

Defines and include files needed to build on the host are host support. Examples are tty support, system defined types, host byte order, host float format.

Defines and information needed to handle the target format are target dependent. Examples are the stack frame format, instruction set, breakpoint instruction, registers, and how to set up and tear down the stack to call a function.

Information that is only needed when the host and target are the same, is native dependent. One example is Unix child process support; if the host and target are not the same, doing a fork to start the target process is a bad idea. The various macros needed for finding the registers in the `upage`, running `ptrace`, and such are all in the native-dependent files.

Another example of native-dependent code is support for features that are really part of the target environment, but which require `#include` files that are only available on the host system. Core file handling and `setjmp` handling are two common cases.

When you want to make GDB work "native" on a particular machine, you have to include all three kinds of information.

3 Algorithms

GDB uses a number of debugging-specific algorithms. They are often not very complicated, but get lost in the thicket of special cases and real-world issues. This chapter describes the basic algorithms and mentions some of the specific target definitions that they use.

3.1 Frames

A frame is a construct that GDB uses to keep track of calling and called functions.

`FRAME_FP` in the machine description has no meaning to the machine-independent part of GDB, except that it is used when setting up a new frame from scratch, as follows:

```
create_new_frame (read_register (FP_REGNUM), read_pc ()));
```

Other than that, all the meaning imparted to `FP_REGNUM` is imparted by the machine-dependent code. So, `FP_REGNUM` can have any value that is convenient for the code that creates new frames. (`create_new_frame` calls `INIT_EXTRA_FRAME_INFO` if it is defined; that is where you should use the `FP_REGNUM` value, if your frames are nonstandard.)

Given a GDB frame, define `FRAME_CHAIN` to determine the address of the calling function's frame. This will be used to create a new GDB frame struct, and then `INIT_EXTRA_FRAME_INFO` and `INIT_FRAME_PC` will be called for the new frame.

3.2 Breakpoint Handling

In general, a breakpoint is a user-designated location in the program where the user wants to regain control if program execution ever reaches that location.

There are two main ways to implement breakpoints; either as "hardware" breakpoints or as "software" breakpoints.

Hardware breakpoints are sometimes available as a builtin debugging features with some chips. Typically these work by having dedicated register into which the breakpoint address may be stored. If the PC ever matches a value in a breakpoint registers, the CPU raises an exception and reports it to GDB. Another possibility is when an emulator is in use; many emulators include circuitry that watches the address lines coming out from the processor, and force it to stop if the address matches a breakpoint's address. A third possibility is that the target already has the ability to do breakpoints somehow; for instance, a ROM monitor may do its own software breakpoints. So although these are not literally "hardware breakpoints", from GDB's point of view they work the same; GDB need not do nothing more than set the breakpoint and wait for something to happen.

Since they depend on hardware resources, hardware breakpoints may be limited in number; when the user asks for more, GDB will start trying to set software breakpoints.

Software breakpoints require GDB to do somewhat more work. The basic theory is that GDB will replace a program instruction a trap, illegal divide, or some other instruction that will cause an exception, and then when it's encountered, GDB will take the exception and stop the program. When the user says to continue, GDB will restore the original instruction, single-step, re-insert the trap, and continue on.

Since it literally overwrites the program being tested, the program area must be writeable, so this technique won't work on programs in ROM. It can also distort the behavior of programs that examine themselves, although the situation would be highly unusual.

Also, the software breakpoint instruction should be the smallest size of instruction, so it doesn't overwrite an instruction that might be a jump target, and cause disaster when the program jumps into the middle of the breakpoint instruction. (Strictly speaking, the breakpoint must be no larger than the smallest interval between instructions that may be jump targets; perhaps there is an architecture where only even-numbered instructions may jumped to.) Note that it's possible for an instruction set not to have any instructions usable for a software breakpoint, although in practice only the ARC has failed to define such an instruction.

The basic definition of the software breakpoint is the macro `BREAKPOINT`.

Basic breakpoint object handling is in '`breakpoint.c`'. However, much of the interesting breakpoint action is in '`infrun.c`'.

3.3 Single Stepping

3.4 Signal Handling

3.5 Thread Handling

3.6 Inferior Function Calls

3.7 Longjmp Support

GDB has support for figuring out that the target is doing a `longjmp` and for stopping at the target of the jump, if we are stepping. This is done with a few specialized internal breakpoints, which are visible in the `maint info breakpoint` command.

To make this work, you need to define a macro called `GET_LONGJMP_TARGET`, which will examine the `jmp_buf` structure and extract the longjmp target address. Since `jmp_buf` is target specific, you will need to define it in the appropriate '`tm-`*xyz*`.h`' file. Look in '`tm-sun4os4.h`' and '`sparc-tdep.c`' for examples of how to do this.

4 User Interface

GDB has several user interfaces. Although the command-line interface is the most common and most familiar, there are others.

4.1 Command Interpreter

The command interpreter in GDB is fairly simple. It is designed to allow for the set of commands to be augmented dynamically, and also has a recursive subcommand capability, where the first argument to a command may itself direct a lookup on a different command list.

For instance, the `set` command just starts a lookup on the `setlist` command list, while `set thread` recurses to the `set_thread_cmd_list`.

To add commands in general, use `add_cmd`. `add_com` adds to the main command list, and should be used for those commands. The usual place to add commands is in the `_initialize_`*xyz* routines at the ends of most source files.

4.2 Console Printing

4.3 TUI

4.4 libgdb

`libgdb` was an abortive project of years ago. The theory was to provide an API to GDB's functionality.

5 Symbol Handling

Symbols are a key part of GDB's operation. Symbols include variables, functions, and types.

5.1 Symbol Reading

GDB reads symbols from "symbol files". The usual symbol file is the file containing the program which GDB is debugging. GDB can be directed to use a different file for symbols (with the `symbol-file` command), and it can also read more symbols via the "add-file" and "load" commands, or while reading symbols from shared libraries.

Symbol files are initially opened by code in '`symfile.c`' using the BFD library. BFD identifies the type of the file by examining its header. `symfile_init` then uses this identification to locate a set of symbol-reading functions.

Symbol reading modules identify themselves to GDB by calling `add_symtab_fns` during their module initialization. The argument to `add_symtab_fns` is a `struct sym_fns` which contains the name (or name prefix) of the symbol format, the length of the prefix, and pointers to four functions. These functions are called at various times to process symbol-files whose identification matches the specified prefix.

The functions supplied by each module are:

xyz`_symfile_init(struct sym_fns *sf)`

> Called from `symbol_file_add` when we are about to read a new symbol file. This function should clean up any internal state (possibly resulting from half-read previous files, for example) and prepare to read a new symbol file. Note that the symbol file which we are reading might be a new "main" symbol file, or might be a secondary symbol file whose symbols are being added to the existing symbol table.
>
> The argument to *xyz*`_symfile_init` is a newly allocated `struct sym_fns` whose `bfd` field contains the BFD for the new symbol file being read. Its `private` field has been zeroed, and can be modified as desired. Typically, a struct of private information will be `malloc`'d, and a pointer to it will be placed in the `private` field.
>
> There is no result from *xyz*`_symfile_init`, but it can call `error` if it detects an unavoidable problem.

xyz`_new_init()`
: Called from `symbol_file_add` when discarding existing symbols. This function need only handle the symbol-reading module's internal state; the symbol table data structures visible to the rest of GDB will be discarded by `symbol_file_add`. It has no arguments and no result. It may be called after *xyz*`_symfile_init`, if a new symbol table is being read, or may be called alone if all symbols are simply being discarded.

xyz`_symfile_read(struct sym_fns *sf, CORE_ADDR addr, int mainline)`
: Called from `symbol_file_add` to actually read the symbols from a symbol-file into a set of psymtabs or symtabs.

  `sf` points to the struct sym_fns originally passed to *xyz*`_sym_init` for possible initialization. `addr` is the offset between the file's specified start address and its true address in memory. `mainline` is 1 if this is the main symbol table being read, and 0 if a secondary symbol file (e.g. shared library or dynamically loaded file) is being read.

In addition, if a symbol-reading module creates psymtabs when *xyz*_symfile_read is called, these psymtabs will contain a pointer to a function *xyz*`_psymtab_to_symtab`, which can be called from any point in the GDB symbol-handling code.

xyz`_psymtab_to_symtab (struct partial_symtab *pst)`
: Called from `psymtab_to_symtab` (or the PSYMTAB_TO_SYMTAB macro) if the psymtab has not already been read in and had its `pst->symtab` pointer set. The argument is the psymtab to be fleshed-out into a symtab. Upon return, pst->readin should have been set to 1, and pst->symtab should contain a pointer to the new corresponding symtab, or zero if there were no symbols in that part of the symbol file.

5.2 Partial Symbol Tables

GDB has three types of symbol tables.

- full symbol tables (symtabs). These contain the main information about symbols and addresses.
- partial symbol tables (psymtabs). These contain enough information to know when to read the corresponding part of the full symbol table.
- minimal symbol tables (msymtabs). These contain information gleaned from non-debugging symbols.

This section describes partial symbol tables.

A psymtab is constructed by doing a very quick pass over an executable file's debugging information. Small amounts of information are extracted – enough to identify which parts of the symbol table will need to be re-read and fully digested later, when the user needs the information. The speed of this pass causes GDB to start up very quickly. Later, as the detailed rereading occurs, it occurs in small pieces, at various times, and the delay therefrom is mostly invisible to the user.

The symbols that show up in a file's psymtab should be, roughly, those visible to the debugger's user when the program is not running code from that file. These include external symbols and types, static symbols and types, and enum values declared at file scope.

The psymtab also contains the range of instruction addresses that the full symbol table would represent.

The idea is that there are only two ways for the user (or much of the code in the debugger) to reference a symbol:

- by its address (e.g. execution stops at some address which is inside a function in this file). The address will be noticed to be in the range of this psymtab, and the full symtab will be read in. `find_pc_function`, `find_pc_line`, and other `find_pc_...` functions handle this.
- by its name (e.g. the user asks to print a variable, or set a breakpoint on a function). Global names and file-scope names will be found in the psymtab, which will cause the symtab to be pulled in. Local names will have to be qualified by a global name, or a file-scope name, in which case we will have already read in the symtab as we evaluated the qualifier. Or, a local symbol can be referenced when we are "in" a local scope, in which case the first case applies. `lookup_symbol` does most of the work here.

The only reason that psymtabs exist is to cause a symtab to be read in at the right moment. Any symbol that can be elided from a psymtab, while still causing that to happen, should not appear in it. Since psymtabs don't have the idea of scope, you can't put local symbols in them anyway. Psymtabs don't have the idea of the type of a symbol, either, so types need not appear, unless they will be referenced by name.

It is a bug for GDB to behave one way when only a psymtab has been read, and another way if the corresponding symtab has been read in. Such bugs are typically caused by a psymtab that does not contain all the visible symbols, or which has the wrong instruction address ranges.

The psymtab for a particular section of a symbol-file (objfile) could be thrown away after the symtab has been read in. The symtab should always be searched before the psymtab, so the psymtab will never be used (in a bug-free environment). Currently, psymtabs are allocated on an obstack, and all the psymbols themselves are allocated in a pair of large arrays on an obstack, so there is little to be gained by trying to free them unless you want to do a lot more work.

5.3 Types

Fundamental Types (e.g., FT_VOID, FT_BOOLEAN).

These are the fundamental types that GDB uses internally. Fundamental types from the various debugging formats (stabs, ELF, etc) are mapped into one of these. They are basically a union of all fundamental types that gdb knows about for all the languages that GDB knows about.

Type Codes (e.g., TYPE_CODE_PTR, TYPE_CODE_ARRAY).

Each time GDB builds an internal type, it marks it with one of these types. The type may be a fundamental type, such as TYPE_CODE_INT, or a derived type, such as TYPE_CODE_PTR which is a pointer to another type. Typically, several FT_* types map to one TYPE_CODE_* type, and are distinguished by other members of the type struct, such as whether the type is signed or unsigned, and how many bits it uses.

Builtin Types (e.g., builtin_type_void, builtin_type_char).

These are instances of type structs that roughly correspond to fundamental types and are created as global types for GDB to use for various ugly historical reasons. We eventually want to eliminate these. Note for example that builtin_type_int initialized in gdbtypes.c is basically the same as a TYPE_CODE_INT type that is initialized in c-lang.c for an FT_INTEGER fundamental type. The difference is that the builtin_type is not associated with any particular objfile, and only one instance exists, while c-lang.c builds as many TYPE_CODE_INT types as needed, with each one associated with some particular objfile.

5.4 Object File Formats

5.4.1 a.out

The '`a.out`' format is the original file format for Unix. It consists of three sections: text, data, and bss, which are for program code, initialized data, and uninitialized data, respectively.

The '`a.out`' format is so simple that it doesn't have any reserved place for debugging information. (Hey, the original Unix hackers used '`adb`', which is a machine-language debugger.) The only debugging format for '`a.out`' is stabs, which is encoded as a set of normal symbols with distinctive attributes.

The basic '`a.out`' reader is in '`dbxread.c`'.

5.4.2 COFF

The COFF format was introduced with System V Release 3 (SVR3) Unix. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited.

The COFF specification includes support for debugging. Although this was a step forward, the debugging information was woefully limited. For instance, it was not possible to represent code that came from an included file.

The COFF reader is in '`coffread.c`'.

5.4.3 ECOFF

ECOFF is an extended COFF originally introduced for Mips and Alpha workstations.

The basic ECOFF reader is in '`mipsread.c`'.

5.4.4 XCOFF

The IBM RS/6000 running AIX uses an object file format called XCOFF. The COFF sections, symbols, and line numbers are used, but debugging symbols are dbx-style stabs whose strings are located in the '`.debug`' section (rather than the string table). For more information, see See section "Top" in *The Stabs Debugging Format*.

The shared library scheme has a clean interface for figuring out what shared libraries are in use, but the catch is that everything which refers to addresses (symbol tables and breakpoints at least) needs to be relocated for both shared libraries and the main executable. At least using the standard mechanism this can only be done once the program has been run (or the core file has been read).

5.4.5 PE

Windows 95 and NT use the PE (Portable Executable) format for their executables. PE is basically COFF with additional headers.

While BFD includes special PE support, GDB needs only the basic COFF reader.

5.4.6 ELF

The ELF format came with System V Release 4 (SVR4) Unix. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations.

The basic ELF reader is in '`elfread.c`'.

5.4.7 SOM

SOM is HP's object file and debug format (not to be confused with IBM's SOM, which is a cross-language ABI).

The SOM reader is in '`hpread.c`'.

5.4.8 Other File Formats

Other file formats that have been supported by GDB include Netware Loadable Modules ('`nlmread.c`'.

5.5 Debugging File Formats

This section describes characteristics of debugging information that are independent of the object file format.

5.5.1 stabs

`stabs` started out as special symbols within the `a.out` format. Since then, it has been encapsulated into other file formats, such as COFF and ELF.

While '`dbxread.c`' does some of the basic stab processing, including for encapsulated versions, '`stabsread.c`' does the real work.

5.5.2 COFF

The basic COFF definition includes debugging information. The level of support is minimal and non-extensible, and is not often used.

5.5.3 Mips debug (Third Eye)

ECOFF includes a definition of a special debug format.

The file '`mdebugread.c`' implements reading for this format.

5.5.4 DWARF 1

DWARF 1 is a debugging format that was originally designed to be used with ELF in SVR4 systems.

The DWARF 1 reader is in '`dwarfread.c`'.

5.5.5 DWARF 2

DWARF 2 is an improved but incompatible version of DWARF 1.

The DWARF 2 reader is in '`dwarf2read.c`'.

5.5.6 SOM

Like COFF, the SOM definition includes debugging information.

5.6 Adding a New Symbol Reader to GDB

If you are using an existing object file format (a.out, COFF, ELF, etc), there is probably little to be done.

If you need to add a new object file format, you must first add it to BFD. This is beyond the scope of this document.

You must then arrange for the BFD code to provide access to the debugging symbols. Generally GDB will have to call swapping routines from BFD and a few other BFD internal routines to locate the debugging information. As much as possible, GDB should not depend on the BFD internal data structures.

For some targets (e.g., COFF), there is a special transfer vector used to call swapping routines, since the external data structures on various platforms have different sizes and layouts. Specialized routines that will only ever be implemented by one object file format may be called directly. This interface should be described in a file '`bfd/libxyz.h`', which is included by GDB.

6 Language Support

GDB's language support is mainly driven by the symbol reader, although it is possible for the user to set the source language manually.

GDB chooses the source language by looking at the extension of the file recorded in the debug info; `.c` means C, `.f` means Fortran, etc. It may also use a special-purpose language identifier if the debug format supports it, such as DWARF.

6.1 Adding a Source Language to GDB

To add other languages to GDB's expression parser, follow the following steps:

Create the expression parser.

This should reside in a file '*lang*`-exp.y`'. Routines for building parsed expressions into a '`union exp_element`' list are in '`parse.c`'.

Since we can't depend upon everyone having Bison, and YACC produces parsers that define a bunch of global names, the following lines *must* be included at the top of the YACC parser, to prevent the various parsers from defining the same global names:

```
#define yyparse  lang_parse
#define yylex  lang_lex
#define yyerror  lang_error
#define yylval  lang_lval
#define yychar  lang_char
#define yydebug  lang_debug
#define yypact   lang_pact
#define yyr1 lang_r1
#define yyr2 lang_r2
#define yydef lang_def
#define yychk lang_chk
#define yypgo lang_pgo
#define yyact   lang_act
#define yyexca   lang_exca
#define yyerrflag   lang_errflag
#define yynerrs   lang_nerrs
```

At the bottom of your parser, define a `struct language_defn` and initialize it with the right values for your language. Define an `initialize_`*lang* routine and have it call '`add_language(`*lang*`_language_defn)`' to tell the rest of GDB that your language exists. You'll need some other supporting variables and functions, which will be used via pointers from your *lang*`_language_defn`. See the declaration of `struct language_defn` in '`language.h`', and the other '`*-exp.y`' files, for more information.

Add any evaluation routines, if necessary

If you need new opcodes (that represent the operations of the language), add them to the enumerated type in '`expression.h`'. Add support code for these operations in `eval.c:evaluate_subexp()`. Add cases for new opcodes in two functions from '`parse.c`': `prefixify_subexp()` and `length_of_subexp()`. These compute the number of `exp_elements` that a given operation takes up.

Update some existing code

Add an enumerated identifier for your language to the enumerated type `enum language` in '`defs.h`'.

Update the routines in '`language.c`' so your language is included. These routines include type predicates and such, which (in some cases) are language dependent. If your language does not appear in the switch statement, an error is reported.

Also included in '`language.c`' is the code that updates the variable `current_language`, and the routines that translate the `language_`*lang* enumerated identifier into a printable string.

Update the function `_initialize_language` to include your language. This function picks the default language upon startup, so is dependent upon which languages that GDB is built for.

Update `allocate_symtab` in '`symfile.c`' and/or symbol-reading code so that the language of each symtab (source file) is set properly. This is used to determine the language to use at each stack frame level. Currently, the language is set based upon the extension of the source file. If the language can be better inferred from the symbol information, please set the language of the symtab in the symbol-reading code.

Add helper code to `expprint.c:print_subexp()` to handle any new expression opcodes you have added to '`expression.h`'. Also, add the printed representations of your operators to `op_print_tab`.

Add a place of call
: Add a call to *lang*`_parse()` and *lang*`_error` in `parse.c:parse_exp_1()`.

Use macros to trim code
: The user has the option of building GDB for some or all of the languages. If the user decides to build GDB for the language *lang*, then every file dependent on '`language.h`' will have the macro `_LANG_`*lang* defined in it. Use `#ifdef`s to leave out large routines that the user won't need if he or she is not using your language.

Note that you do not need to do this in your YACC parser, since if GDB is not build for *lang*, then '*lang*`-exp.tab.o`' (the compiled form of your parser) is not linked into GDB at all.

See the file '`configure.in`' for how GDB is configured for different languages.

Edit '`Makefile.in`'
: Add dependencies in '`Makefile.in`'. Make sure you update the macro variables such as `HFILES` and `OBJS`, otherwise your code may not get linked in, or, worse yet, it may not get `tar`red into the distribution!

7 Host Definition

With the advent of autoconf, it's rarely necessary to have host definition machinery anymore.

7.1 Adding a New Host

Most of GDB's host configuration support happens via autoconf. It should be rare to need new host-specific definitions. GDB still uses the host-specific definitions and files listed below, but these mostly exist for historical reasons, and should eventually disappear.

Several files control GDB's configuration for host systems:

'`gdb/config/`*arch*`/`*xyz*`.mh`'
: Specifies Makefile fragments needed when hosting on machine *xyz*. In particular, this lists the required machine-dependent object files, by defining '`XDEPFILES=...`'. Also specifies the header file which describes host *xyz*, by defining `XM_FILE= xm-`*xyz*`.h`. You can also define `CC`, `SYSV_DEFINE`, `XM_CFLAGS`, `XM_ADD_FILES`, `XM_CLIBS`, `XM_CDEPS`, etc.; see '`Makefile.in`'.

'gdb/config/*arch*/xm-*xyz*.h'
: ('xm.h' is a link to this file, created by configure). Contains C macro definitions describing the host system environment, such as byte order, host C compiler and library.

'gdb/*xyz*-xdep.c'
: Contains any miscellaneous C code required for this machine as a host. On most machines it doesn't exist at all. If it does exist, put '*xyz*-xdep.o' into the XDEPFILES line in 'gdb/config/*arch*/*xyz*.mh'.

Generic Host Support Files

There are some "generic" versions of routines that can be used by various systems. These can be customized in various ways by macros defined in your 'xm-*xyz*.h' file. If these routines work for the *xyz* host, you can just include the generic file's name (with '.o', not '.c') in XDEPFILES.

Otherwise, if your machine needs custom support routines, you will need to write routines that perform the same functions as the generic file. Put them into *xyz*-xdep.c, and put *xyz*-xdep.o into XDEPFILES.

'ser-unix.c'
: This contains serial line support for Unix systems. This is always included, via the makefile variable SER_HARDWIRE; override this variable in the '.mh' file to avoid it.

'ser-go32.c'
: This contains serial line support for 32-bit programs running under DOS, using the GO32 execution environment.

'ser-tcp.c'
: This contains generic TCP support using sockets.

7.2 Host Conditionals

When GDB is configured and compiled, various macros are defined or left undefined, to control compilation based on the attributes of the host system. These macros and their meanings (or if the meaning is not documented here, then one of the source files where they are used is indicated) are:

GDBINIT_FILENAME
: The default name of GDB's initialization file (normally '.gdbinit').

MEM_FNS_DECLARED
: Your host config file defines this if it includes declarations of memcpy and memset. Define this to avoid conflicts between the native include files and the declarations in 'defs.h'.

NO_STD_REGS
: This macro is deprecated.

NO_SYS_FILE
: Define this if your system does not have a <sys/file.h>.

`SIGWINCH_HANDLER`
: If your host defines `SIGWINCH`, you can define this to be the name of a function to be called if `SIGWINCH` is received.

`SIGWINCH_HANDLER_BODY`
: Define this to expand into code that will define the function named by the expansion of `SIGWINCH_HANDLER`.

`ALIGN_STACK_ON_STARTUP`
: Define this if your system is of a sort that will crash in `tgetent` if the stack happens not to be longword-aligned when `main` is called. This is a rare situation, but is known to occur on several different types of systems.

`CRLF_SOURCE_FILES`
: Define this if host files use `\r\n` rather than `\n` as a line terminator. This will cause source file listings to omit `\r` characters when printing and it will allow \r\n line endings of files which are "sourced" by gdb. It must be possible to open files in binary mode using `O_BINARY` or, for fopen, `"rb"`.

`DEFAULT_PROMPT`
: The default value of the prompt string (normally `"(gdb) "`).

`DEV_TTY` The name of the generic TTY device, defaults to `"/dev/tty"`.

`FCLOSE_PROVIDED`
: Define this if the system declares `fclose` in the headers included in `defs.h`. This isn't needed unless your compiler is unusually anal.

`FOPEN_RB` Define this if binary files are opened the same way as text files.

`GETENV_PROVIDED`
: Define this if the system declares `getenv` in its headers included in `defs.h`. This isn't needed unless your compiler is unusually anal.

`HAVE_MMAP`
: In some cases, use the system call `mmap` for reading symbol tables. For some machines this allows for sharing and quick updates.

`HAVE_SIGSETMASK`
: Define this if the host system has job control, but does not define `sigsetmask()`. Currently, this is only true of the RS/6000.

`HAVE_TERMIO`
: Define this if the host system has `termio.h`.

`HOST_BYTE_ORDER`
: The ordering of bytes in the host. This must be defined to be either `BIG_ENDIAN` or `LITTLE_ENDIAN`.

`INT_MAX`

`INT_MIN`

`LONG_MAX`

`UINT_MAX`

`ULONG_MAX`
: Values for host-side constants.

`ISATTY`
: Substitute for isatty, if not available.

`LONGEST`
: This is the longest integer type available on the host. If not defined, it will default to `long long` or `long`, depending on `CC_HAS_LONG_LONG`.

`CC_HAS_LONG_LONG`
: Define this if the host C compiler supports "long long". This is set by the configure script.

`PRINTF_HAS_LONG_LONG`
: Define this if the host can handle printing of long long integers via the printf format directive "ll". This is set by the configure script.

`HAVE_LONG_DOUBLE`
: Define this if the host C compiler supports "long double". This is set by the configure script.

`PRINTF_HAS_LONG_DOUBLE`
: Define this if the host can handle printing of long double float-point numbers via the printf format directive "Lg". This is set by the configure script.

`SCANF_HAS_LONG_DOUBLE`
: Define this if the host can handle the parsing of long double float-point numbers via the scanf format directive directive "Lg". This is set by the configure script.

`LSEEK_NOT_LINEAR`
: Define this if `lseek (n)` does not necessarily move to byte number `n` in the file. This is only used when reading source files. It is normally faster to define `CRLF_SOURCE_FILES` when possible.

`L_SET`
: This macro is used as the argument to lseek (or, most commonly, bfd_seek). FIXME, should be replaced by SEEK_SET instead, which is the POSIX equivalent.

`MALLOC_INCOMPATIBLE`
: Define this if the system's prototype for `malloc` differs from the ANSI definition.

`MMAP_BASE_ADDRESS`
: When using HAVE_MMAP, the first mapping should go at this address.

`MMAP_INCREMENT`
: when using HAVE_MMAP, this is the increment between mappings.

`NEED_POSIX_SETPGID`
: Define this to use the POSIX version of `setpgid` to determine whether job control is available.

`NORETURN`
: If defined, this should be one or more tokens, such as `volatile`, that can be used in both the declaration and definition of functions to indicate that they never return. The default is already set correctly if compiling with GCC. This will almost never need to be defined.

`ATTR_NORETURN`
: If defined, this should be one or more tokens, such as `__attribute__ ((noreturn))`, that can be used in the declarations of functions to indicate that they never return. The default is already set correctly if compiling with GCC. This will almost never need to be defined.

`USE_GENERIC_DUMMY_FRAMES`
: Define this to 1 if the target is using the generic inferior function call code. See `blockframe.c` for more information.

`USE_MMALLOC`
: GDB will use the `mmalloc` library for memory allocation for symbol reading if this symbol is defined. Be careful defining it since there are systems on which `mmalloc` does not work for some reason. One example is the DECstation, where its RPC library can't cope with our redefinition of `malloc` to call `mmalloc`. When defining `USE_MMALLOC`, you will also have to set `MMALLOC` in the Makefile, to point to the mmalloc library. This define is set when you configure with –with-mmalloc.

`NO_MMCHECK`
: Define this if you are using `mmalloc`, but don't want the overhead of checking the heap with `mmcheck`. Note that on some systems, the C runtime makes calls to malloc prior to calling `main`, and if `free` is ever called with these pointers after calling `mmcheck` to enable checking, a memory corruption abort is certain to occur. These systems can still use mmalloc, but must define NO_MMCHECK.

`MMCHECK_FORCE`
: Define this to 1 if the C runtime allocates memory prior to `mmcheck` being called, but that memory is never freed so we don't have to worry about it triggering a memory corruption abort. The default is 0, which means that `mmcheck` will only install the heap checking functions if there has not yet been any memory allocation calls, and if it fails to install the functions, gdb will issue a warning. This is currently defined if you configure using –with-mmalloc.

`NO_SIGINTERRUPT`
: Define this to indicate that siginterrupt() is not available.

`R_OK`
: Define if this is not in a system .h file.

`SEEK_CUR`

`SEEK_SET`
: Define these to appropriate value for the system lseek(), if not already defined.

`STOP_SIGNAL`
: This is the signal for stopping GDB. Defaults to SIGTSTP. (Only redefined for the Convex.)

`USE_O_NOCTTY`
: Define this if the interior's tty should be opened with the O_NOCTTY flag. (FIXME: This should be a native-only flag, but '`inflow.c`' is always linked in.)

`USG`
: Means that System V (prior to SVR4) include files are in use. (FIXME: This symbol is abused in '`infrun.c`', '`regex.c`', '`remote-nindy.c`', and '`utils.c`' for other things, at the moment.)

`lint` Define this to help placate lint in some situations.

`volatile` Define this to override the defaults of `__volatile__` or `/**/`.

8 GDB Overview

8.1 Libraries used by GDB

GDB relies on a number of libraries:

'`config/`' Configuration options shared by GDB and all of the libraries it uses. GDB has it's own much more extensive configuration in '`gdb/config`'.

'`readline/`'

Contains the `-lreadline` and `-lhistory` libraries for command-line processing. The `-lreadline` library handles command-line editing, terminal interface, keymap interfaces, and file completion; the `-lhistory` library handles history processing and history substitution using csh-style syntax. For more information, see '`readline/doc/hist.texi`' and '`readline/doc/rlman.texi`'.

'`bfd/`' BFD is a package which allows applications to use the same routines to operate on object files whatever the object file format. A new object file format can be supported simply by creating a new BFD back end and adding it to the library.

BFD is split into two parts: the front end, and the back ends (one for each object file format).

- The front end of BFD provides the interface to the user. It manages memory and various canonical data structures. The front end also decides which back end to use and when to call back end routines.
- The back ends provide BFD its view of the real world. Each back end provides a set of calls which the BFD front end can use to maintain its canonical form. The back ends also may keep around information for their own use, for greater efficiency.

For more information on BFD, see '`bfd/doc/bfd.texi`'. For more information on the use of BFD in GDB, see ⟨undefined⟩ [BFD support for GDB], page ⟨undefined⟩.

'`opcodes/`'

A collection of routines to parse/print machine-language instructions and arguments for a number of processors.

'`mmalloc/`'

The gnu `malloc()` library.

'`sim/`' Simulators for a number of common microprocessors. Allows GDB to debug machine code for architectures for which no CPU is readily available or which are not yet capable of interacting with GDB directly. Generally used to simulate and/or debug embedded systems.

'texinfo/'
: Texinfo is a documentation system that uses a single source file to produce both online information and printed output. Most GDB-related documentation is produced using this system. For more information, see 'texinfo/doc/texinfo.texi'.

'libiberty/'
: Contains the -liberty library of free software. It is a collection of subroutines used by various GNU programs, typically functions that are included in GNU libc, but not in certain vendor versions of libc. Example functions provided by -liberty:

getopt	get options from command line
obstack	stacks of arbitrarily-sized objects
strerror	error message strings corresponding to errno
strtol	string-to-long conversion
strtoul	string-to-unsigned-long conversion

8.2 GDB Directory Structure

The sources to GDB itself are currently stored in four subdirectories, all of which are used to build the final executable:

'gdb/'
: The main GDB sources as provided by the FSF. Changes to this directory have been kept as small as possible to minimize the effort of merging them with the FSF sources (though many changes still have been made).

'gdb-next/'
: Apple-specific additions to the GDB source base. These will eventually be merged into the 'gdb/' directory, but are currently kept separate to make them easier to manage.

'gdb-4.14/'
: GDB sources as modified by NeXT for gdb-4.14. These are nearly obsolete, but are kept around until I have merged all of the original NeXT changes into the new GDB source base.

'gdb-next-4.14/'
: Apple-specific additions to the (nearly obsolete) GDB 4.14 source base. These are kept around solely for the purpose of building GDB 4.14 binaries when necessary.

Until recently, it was possible to build both GDB 4.17 and the GDB 4.14/4.17 hybrid that shipped with DR1 from the same source tree. GDB 4.17 was built in 'gdb' and used files from 'gdb-next' and 'gdb/', in that order, and GDB 4.14 was built in 'gdb-4.14' and used the files from 'gdb-next-4.14/', 'gdb-next/', 'gdb-4.14/', and 'gdb/', also in that order.

As of January 8, I have stopped building GDB 4.14 along with GDB 4.17 from the same sources. I suspect GDB 4.14 will no longer build from these sources without modification, although I suspect the necessary changes would be relatively minor.

GDB also uses the following subdirectories:

'include/'
: Header files shared by GDB and all of the libraries it uses. These files typically will typically apply to either:

  General operating-system functions (such as symbol reading, IEEE floats, etc.) for which GDB and it's libraries want to have a common interface.

  Interfaces between GDB and one of its libraries (such as 'remote-sim.h' and 'libiberty.h').

'gdb/config/'
: Contains GDB-specific configuration files. For more information see ⟨undefined⟩ [Config], page ⟨undefined⟩.

8.3 Overview of Source Files

8.3.1 Top Level

'main.c'
: Contains the GDB top-level interpreter. Parses command-line arguments, performs GDB initialization, and passes control to command-line interpreter.

'maint.c'
: Collection of utility functions for viewing/debugging the internal state of GDB.

'top.c'
: Top-level routine used by GDB. Evaluates commands, and provides a number of top-level functions and mechanisms to set general purpose variables.

'command.c'
: parse and evaluate gdb commands and command documentation

'printcmd.c'
: user commands for printing expressions and displaying memory

8.3.2 GDB Targets / Program Control

'target.c'
: Defines the target abstraction, used to encapsulate all communications between GDB and a target. See the comments in 'target.h' for more information.

'exec.c'
: Interface between executable files and the 'target' abstraction. Allows GDB to inspect/analyze executable images, without necessarily having a debuggable process available.

'corefile.c'
: Interface between core images and the 'target' abstraction. Allows GDB to inspect/analyze memory and register dumps from corefile images. Not used in GDB 4.14.

'thread.c'
: Interface between GDB and multiply-threaded subprocesses. Not currently used by either GDB 4.14 or GDB 4.17.

'fork-child.c'
: Code to create an inferior process on UNIX systems.

'infcmd.c'
: User-level commands for inspecting/controlling the state of process execution. Commands such as 'step', 'next', 'finish', and 'continue' go here.

'inflow.c'
: Handles terminal modes and signal handling for UNIX child processes.

'infrun.c'
: Target-independent code to control the execution of an inferior process. Handles breakpoints, signal handling, shared library handling (on some systems), as well as far too many other things. Contains `wait_for_inferior`, probably the hairiest function in all of GDB.

'inftarg.c'
: Interface between the GDB 'target' abstraction and UNIX child processes. Many of the functions in the file are overridden (via macros) in the `nextstep-*` functions.

8.3.3 Types, Values, and Expressions

A `type` is the fundamental data structure in GDB for representing type information. Each `type` structure is associated with a particular object file, with the exception of a few pre-created type structures used for backwards compatibility with other parts of GDB. GDB provides a number of "fundamental" data types; more complex data types can be represented by nesting `type` structures within each other. See Section 8.3.3 [Types], page 20 for more information.

A `value` is the GDB data structure for representing both R- and L-values of any type. A `value` contains a pointer to a GDB `type` structure, as well as a region of memory containing the value's contents (for an R-value) or address (for an L-value).

A `expression` is the GDB data structure for all expressions in all programming languages. Expressions can be parsed and evaluated interactively according to the current language syntax, can be used by breakpoints and watchpoints to compute values, and can cause execution to take place within a target process (by evaluating function expressions). Expressions are parsed, evaluated, and printed using the language-dispatching mechanisms described in '`language.c`' and Section 8.3.7 [Language-Specific Sources], page 23.

GDB source files to manipulate `type` structures:

'typeprint.c'
: Prints type information structures in readable form. Interfaces to the language-specific type printing routines described in Section 8.3.7 [Language-Specific Sources], page 23.

'gdbtypes.c'
: Fundamental type definitions used by GDB.

GDB source files to manipulate `expression` structures:

'parse.c'
: Parse expressions typed at the command-line into `expression`. Interfaces to the language-specific expression parsing routines described in Section 8.3.7 [Language-Specific Sources], page 23.

'eval.c' Evaluates `expression` structures in the current execution context.

'expprint.c'
: Prints `expression` structures in readable (infix) form. Interfaces to the language-specific type printing routines described in Section 8.3.7 [Language-Specific Sources], page 23.

GDB source files to manipulate `value` structures:

'valarith.c'
: perform arithmetic operations on values

'valops.c'
: perform non-arithmetic operations on values

'valprint.c'
: print functions for values

'values.c'
: low-level packing/unpacking of value objects to/from raw format

'findvar.c'
: resolve variables to their value structures (as appropriate to the current stack environment).

8.3.4 Stack Analysis

'blockframe.c'
: machine-independent code to analyze stack frames

'stack.c' machine-independent stack frame analysis, user-level commands to manage and inspect the stack.

8.3.5 Breakpoints

'breakpoint.c'
: Machine-independent breakpoint code. Handles and dispatches all forms of breakpoints, including hardware breakpoints, software breakpoints, hardware and software watchpoints, and shared library breakpoints. Contains top-level commands to set, inspect, and manipulate breakpoints and watchpoints. Provides routine to read memory from inferior, replacing any memory that may have been overwritten by a breakpoint with its saved value.

'mem-break.c'
: Implements software breakpoints. To set a breakpoint at a given location, GDB saves the instruction at that location and inserts a software trap instruction in its place.

8.3.6 Symbol File Management

The following files allow GDB to parse and manage symbol information in a variety of formats. For an overview of GDB object file and symbol handling, see ⟨undefined⟩ [Symbol Tables], page ⟨undefined⟩.

'objfiles.c'
: create/destroy/manage objfile structures

'symfile.c'
: top-level commands to handle objfiles, also handles overlays (unused)

'symmisc.c'
: various objfile and (p)symtab utilities (mainly debugging)

'symtab.c'
: basic symbol table utilities (mainly lookup)

'minsyms.c'
: manage minimal symbol tables

'buildsym.c'
: build complete symbol data structures

The following source files provide symbol-reading interfaces for various file formats. Although all these files are compiled into GDB for Mac OS X, only the first three ('stabsread', 'dbxread', and 'machoread') are actively used by the rest of GDB.

'stabsread.c'
: common stabs parsing routines

'dbxread.c'
: read DBX (stabs) symbol files

'machoread.c'
: read Mach-O (stabs) symbol files

'nlmread.c'
: read Netware NLM symbol files (unused)

'os9kread.c'
: read OS9/OS9K symbol files (unused)

'dwarf2read.c'
: read DWARF2 symbol files (unused)

'dwarfread.c'
: read DWARF symbol files (unused)

'elfread.c'
: read ELF symbol files (unused)

'coffread.c'
: read COFF symbol files (unused)

'mdebugread.c'
: read ECOFF symbol files (unused)

'mipsread.c'
: read MIPS symbol files (unused)

8.3.7 Language-Specific Sources

The following files are used to provide language-specific expression evaluation and printing support. The file *lang*-exp handles expression parsing, *lang*-typeprint prints human-readable versions of GDB 'type' structures, and *lang*-valprint prints human-readable versions of GDB 'value' structures, and *lang*-lang provides general language-specific support functions. For more information on language-specific support in GDB, see ⟨undefined⟩ [Languages], page ⟨undefined⟩.

`C` 'c-exp', 'c-lang', 'c-typeprint', 'c-valprint'

`C++` 'cp-valprint'

`Objective-C`
: 'objc-exp', 'objc-lang', 'objc'

`Chill` 'ch-exp', 'ch-lang', 'ch-typeprint', 'ch-valprint'

`Fortran` 'f-exp', 'f-lang', 'f-typeprint', 'f-valprint'

`Java` 'jv-exp', 'jv-lang', 'jv-typeprint', 'jv-valprint'

`Modula II` 'm2-exp', 'm2-lang', 'm2-typeprint', 'm2-valprint'

`Scheme` 'scm-exp', 'scm-lang', 'scm-valprint'

8.3.8 Kernel Debugging

'remote-kdp.c'
: interface gdb 'target' interface to a remote Mac OS X kernel via KDP

'kdp-udp.c'
: communications library for KDP

'remote-utils.c'
: more serial-line support (unused)

'remote.c'
: generic serial-line debugging (unused)

'ser-tcp.c'
: treat TCP connection as serial line (unused)

'ser-unix.c'
: interface to unix serial ports (unused)

'serial.c'
: implement generic serial routines (unused)

8.3.9 Sources Specific to Mac OS X

'nextstep-nat-dyld.c'
: handle dyld debugging messages and take action (mainly load symfiles)

'nextstep-nat-inferior.c'
: interface between GDB 'target' abstraction and Mach process control functions

'nextstep-nat-mutils.c'
: mach functions to read/write memory, other manipulations

'nextstep-nat-sigthread.c'
: create/manage thread to detect signals sent to the inferior process

'nextstep-nat-threads.c'
: interface gdb 'target' interface to a running Mac OS X process on the same machine

'nextstep-tdep.c'
: extra functions needed for all nextstep targets (empty)

'nextstep-xdep.c'
: extra functions needed for nextstep hosts (empty)

8.3.10 PowerPC-specific Sources

'ppc-frameinfo.c'
: determine/print PPC stack frame info (find prologue, etc)

'ppc-frameops.c'
: basic PPC stack frame operations (push dummy, push args, pop, find saved regs)

'ppc-next-tcore.c'
: fetch/store PPC registers to/from running Mach thread

'ppc-next-tdep.c'
: fetch/store PPC registers from Mach data structure

'ppc-tdep.c'
: PPC analysis functions used by rest of GDB

8.3.11 Miscellaneous

'demangle'
: Allow user to explicitly select function name demangling style (e.g. 'k+r', 'lucid', 'GNU').

'language'
: Multiple-language support for GDB. Contains dispatch tables to the language-specific routines, routines to detect the language of a source file / object module, and explicit commands to print/set the current language environment.

'source'
: View and navigate through source files. Also responsible for directory search path mechanisms.

8.3.12 Assorted Utilities

'bcache'
: efficiency additions for obstack

'gnu-regex'
: regular expression library

'dcache' caches inferior memory accesses (for remote targets)

'complaints'
error-printing for symbol file reading allows error messages to be printed only once per operation, rather than repeated incessantly

'copying' prints license/warranty information

'version' automatically generated, contains version string

'annotate'
convenience functions to print annotations for libgdb

'environ' utilities to read/modify the process environment

'utils' Utilities used by GDB. Provides routines to provide formatted output, memory management, and data conversion routines.

9 Target Architecture Definition

GDB's target architecture defines what sort of machine-language programs GDB can work with, and how it works with them.

At present, the target architecture definition consists of a number of C macros.

9.1 Registers and Memory

GDB's model of the target machine is rather simple. GDB assumes the machine includes a bank of registers and a block of memory. Each register may have a different size.

GDB does not have a magical way to match up with the compiler's idea of which registers are which; however, it is critical that they do match up accurately. The only way to make this work is to get accurate information about the order that the compiler uses, and to reflect that in the `REGISTER_NAME` and related macros.

GDB can handle big-endian, little-endian, and bi-endian architectures.

9.2 Frame Interpretation

9.3 Inferior Call Setup

9.4 Compiler Characteristics

9.5 Target Conditionals

This section describes the macros that you can use to define the target machine.

`ADDITIONAL_OPTIONS`

`ADDITIONAL_OPTION_CASES`

`ADDITIONAL_OPTION_HANDLER`

`ADDITIONAL_OPTION_HELP`
: These are a set of macros that allow the addition of additional command line options to GDB. They are currently used only for the unsupported i960 Nindy target, and should not be used in any other configuration.

`ADDR_BITS_REMOVE (addr)`
: If a raw machine address includes any bits that are not really part of the address, then define this macro to expand into an expression that zeros those bits in *addr*. For example, the two low-order bits of a Motorola 88K address may be used by some kernels for their own purposes, since addresses must always be 4-byte aligned, and so are of no use for addressing. Those bits should be filtered out with an expression such as `((addr) & ~3)`.

`BEFORE_MAIN_LOOP_HOOK`
: Define this to expand into any code that you want to execute before the main loop starts. Although this is not, strictly speaking, a target conditional, that is how it is currently being used. Note that if a configuration were to define it one way for a host and a different way for the target, GDB will probably not compile, let alone run correctly. This is currently used only for the unsupported i960 Nindy target, and should not be used in any other configuration.

`BELIEVE_PCC_PROMOTION`
: Define if the compiler promotes a short or char parameter to an int, but still reports the parameter as its original type, rather than the promoted type.

`BELIEVE_PCC_PROMOTION_TYPE`
: Define this if GDB should believe the type of a short argument when compiled by pcc, but look within a full int space to get its value. Only defined for Sun-3 at present.

`BITS_BIG_ENDIAN`
: Define this if the numbering of bits in the targets does *not* match the endianness of the target byte order. A value of 1 means that the bits are numbered in a big-endian order, 0 means little-endian.

`BREAKPOINT`
: This is the character array initializer for the bit pattern to put into memory where a breakpoint is set. Although it's common to use a trap instruction for a breakpoint, it's not required; for instance, the bit pattern could be an invalid instruction. The breakpoint must be no longer than the shortest instruction of the architecture.

: *BREAKPOINT* has been deprecated in favour of *BREAKPOINT_FROM_PC*.

`BIG_BREAKPOINT`

`LITTLE_BREAKPOINT`
: Similar to BREAKPOINT, but used for bi-endian targets.

  BIG_BREAKPOINT and *LITTLE_BREAKPOINT* have been deprecated in favour of *BREAKPOINT_FROM_PC*.

`REMOTE_BREAKPOINT`

`LITTLE_REMOTE_BREAKPOINT`

`BIG_REMOTE_BREAKPOINT`
: Similar to BREAKPOINT, but used for remote targets.

  BIG_REMOTE_BREAKPOINT and *LITTLE_REMOTE_BREAKPOINT* have been deprecated in favour of *BREAKPOINT_FROM_PC*.

`BREAKPOINT_FROM_PC (pcptr, lenptr)`
: Use the program counter to determine the contents and size of a breakpoint instruction. It returns a pointer to a string of bytes that encode a breakpoint instruction, stores the length of the string to *lenptr, and adjusts pc (if necessary) to point to the actual memory location where the breakpoint should be inserted.

  Although it is common to use a trap instruction for a breakpoint, it's not required; for instance, the bit pattern could be an invalid instruction. The breakpoint must be no longer than the shortest instruction of the architecture.

  Replaces all the other *BREAKPOINT* macros.

`CALL_DUMMY_P`
: A C expresson that is non-zero when the target suports inferior function calls.

`CALL_DUMMY_WORDS`
: Pointer to an array of *LONGEST* words of data containing host-byte-ordered *REGISTER_BYTES* sized values that partially specify the sequence of instructions needed for an inferior function call.

  Should be deprecated in favour of a macro that uses target-byte-ordered data.

`SIZEOF_CALL_DUMMY_WORDS`
: The size of *CALL_DUMMY_WORDS*. When *CALL_DUMMY_P* this must return a positive value. See also *CALL_DUMMY_LENGTH*.

`CALL_DUMMY`
: A static initializer for *CALL_DUMMY_WORDS*. Deprecated.

`CALL_DUMMY_LOCATION`
: inferior.h

`CALL_DUMMY_STACK_ADJUST`
: Stack adjustment needed when performing an inferior function call.

  Should be deprecated in favor of something like *STACK_ALIGN*.

`CALL_DUMMY_STACK_ADJUST_P`
: Predicate for use of *CALL_DUMMY_STACK_ADJUST*.

  Should be deprecated in favor of something like *STACK_ALIGN*.

`CANNOT_FETCH_REGISTER (regno)`
: A C expression that should be nonzero if *regno* cannot be fetched from an inferior process. This is only relevant if `FETCH_INFERIOR_REGISTERS` is not defined.

`CANNOT_STORE_REGISTER (regno)`
: A C expression that should be nonzero if *regno* should not be written to the target. This is often the case for program counters, status words, and other special registers. If this is not defined, GDB will assume that all registers may be written.

`DO_DEFERRED_STORES`
`CLEAR_DEFERRED_STORES`
: Define this to execute any deferred stores of registers into the inferior, and to cancel any deferred stores.

  Currently only implemented correctly for native Sparc configurations?

`CPLUS_MARKER`
: Define this to expand into the character that G++ uses to distinguish compiler-generated identifiers from programmer-specified identifiers. By default, this expands into `'$'`. Most System V targets should define this to `'.'`.

`DBX_PARM_SYMBOL_CLASS`
: Hook for the `SYMBOL_CLASS` of a parameter when decoding DBX symbol information. In the i960, parameters can be stored as locals or as args, depending on the type of the debug record.

`DECR_PC_AFTER_BREAK`
: Define this to be the amount by which to decrement the PC after the program encounters a breakpoint. This is often the number of bytes in BREAKPOINT, though not always. For most targets this value will be 0.

`DECR_PC_AFTER_HW_BREAK`
: Similarly, for hardware breakpoints.

`DISABLE_UNSETTABLE_BREAK addr`
: If defined, this should evaluate to 1 if *addr* is in a shared library in which breakpoints cannot be set and so should be disabled.

`DO_REGISTERS_INFO`
: If defined, use this to print the value of a register or all registers.

`END_OF_TEXT_DEFAULT`
: This is an expression that should designate the end of the text section (? FIXME ?)

`EXTRACT_RETURN_VALUE(type,regbuf,valbuf)`
: Define this to extract a function's return value of type *type* from the raw register state *regbuf* and copy that, in virtual format, into *valbuf*.

`EXTRACT_STRUCT_VALUE_ADDRESS(regbuf)`
: When *EXTRACT_STRUCT_VALUE_ADDRESS_P* this is used to to extract from an array *regbuf* (containing the raw register state) the address in which a

function should return its structure value, as a CORE_ADDR (or an expression that can be used as one).

`EXTRACT_STRUCT_VALUE_ADDRESS_P`
: Predicate for *EXTRACT_STRUCT_VALUE_ADDRESS*.

`FLOAT_INFO`
: If defined, then the 'info float' command will print information about the processor's floating point unit.

`FP_REGNUM`
: If the virtual frame pointer is kept in a register, then define this macro to be the number (greater than or equal to zero) of that register.

  This should only need to be defined if `TARGET_READ_FP` and `TARGET_WRITE_FP` are not defined.

`FRAMELESS_FUNCTION_INVOCATION(fi)`
: Define this to an expression that returns 1 if the function invocation represented by *fi* does not have a stack frame associated with it. Otherwise return 0.

`FRAME_ARGS_ADDRESS_CORRECT`
: stack.c

`FRAME_CHAIN(frame)`
: Given *frame*, return a pointer to the calling frame.

`FRAME_CHAIN_COMBINE(chain,frame)`
: Define this to take the frame chain pointer and the frame's nominal address and produce the nominal address of the caller's frame. Presently only defined for HP PA.

`FRAME_CHAIN_VALID(chain,thisframe)`
: Define this to be an expression that returns zero if the given frame is an outermost frame, with no caller, and nonzero otherwise. Three common definitions are available. `default_frame_chain_valid` (the default) is nonzero if the chain pointer is nonzero and given frame's PC is not inside the startup file (such as '`crt0.o`'). `alternate_frame_chain_valid` is nonzero if the chain pointer is nonzero and the given frame's PC is not in `main()` or a known entry point function (such as `_start()`).

`FRAME_INIT_SAVED_REGS(frame)`
: See '`frame.h`'. Determines the address of all registers in the current stack frame storing each in `frame->saved_regs`. Space for `frame->saved_regs` shall be allocated by `FRAME_INIT_SAVED_REGS` using either `frame_saved_regs_zalloc` or `frame_obstack_alloc`.

  FRAME_FIND_SAVED_REGS and *EXTRA_FRAME_INFO* are deprecated.

`FRAME_NUM_ARGS (fi)`
: For the frame described by *fi* return the number of arguments that are being passed. If the number of arguments is not known, return `-1`.

`FRAME_SAVED_PC(frame)`
: Given *frame*, return the pc saved there. That is, the return address.

`FUNCTION_EPILOGUE_SIZE`
: For some COFF targets, the `x_sym.x_misc.x_fsize` field of the function end symbol is 0. For such targets, you must define `FUNCTION_EPILOGUE_SIZE` to expand into the standard size of a function's epilogue.

`GCC_COMPILED_FLAG_SYMBOL`

`GCC2_COMPILED_FLAG_SYMBOL`
: If defined, these are the names of the symbols that GDB will look for to detect that GCC compiled the file. The default symbols are `gcc_compiled.` and `gcc2_compiled.`, respectively. (Currently only defined for the Delta 68.)

`GDB_MULTI_ARCH`
: If defined and non-zero, enables suport for multiple architectures within GDB.

  The support can be enabled at two levels. At level one, only definitions for previously undefined macros are provided; at level two, a multi-arch definition of all architecture dependant macros will be defined.

`GDB_TARGET_IS_HPPA`
: This determines whether horrible kludge code in dbxread.c and partial-stab.h is used to mangle multiple-symbol-table files from HPPA's. This should all be ripped out, and a scheme like elfread.c used.

`GDB_TARGET_IS_MACH386`

`GDB_TARGET_IS_SUN3`

`GDB_TARGET_IS_SUN386`
: Kludges that should go away.

`GET_LONGJMP_TARGET`
: For most machines, this is a target-dependent parameter. On the DECstation and the Iris, this is a native-dependent parameter, since `<setjmp.h>` is needed to define it.

  This macro determines the target PC address that longjmp() will jump to, assuming that we have just stopped at a longjmp breakpoint. It takes a CORE_ADDR * as argument, and stores the target PC value through this pointer. It examines the current state of the machine as needed.

`GET_SAVED_REGISTER`
: Define this if you need to supply your own definition for the function `get_saved_register`.

`HAVE_REGISTER_WINDOWS`
: Define this if the target has register windows.

`REGISTER_IN_WINDOW_P (regnum)`
: Define this to be an expression that is 1 if the given register is in the window.

`IBM6000_TARGET`
: Shows that we are configured for an IBM RS/6000 target. This conditional should be eliminated (FIXME) and replaced by feature-specific macros. It was introduced in haste and we are repenting at leisure.

`IEEE_FLOAT`
: Define this if the target system uses IEEE-format floating point numbers.

`INIT_EXTRA_FRAME_INFO (fromleaf, frame)`
: If additional information about the frame is required this should be stored in `frame->extra_info`. Space for `frame->extra_info` is allocated using `frame_obstack_alloc`.

`INIT_FRAME_PC (fromleaf, prev)`
: This is a C statement that sets the pc of the frame pointed to by *prev*. [By default...]

`INNER_THAN (lhs,rhs)`
: Returns non-zero if stack address *lhs* is inner than (nearer to the stack top) stack address *rhs*. Define this as `lhs < rhs` if the target's stack grows downward in memory, or `lhs > rsh` if the stack grows upward.

`IN_SIGTRAMP (pc, name)`
: Define this to return true if the given *pc* and/or *name* indicates that the current function is a sigtramp.

`SIGTRAMP_START (pc)`

`SIGTRAMP_END (pc)`
: Define these to be the start and end address of the sigtramp for the given *pc*. On machines where the address is just a compile time constant, the macro expansion will typically just ignore the supplied *pc*.

`IN_SOLIB_CALL_TRAMPOLINE pc name`
: Define this to evaluate to nonzero if the program is stopped in the trampoline that connects to a shared library.

`IN_SOLIB_RETURN_TRAMPOLINE pc name`
: Define this to evaluate to nonzero if the program is stopped in the trampoline that returns from a shared library.

`IS_TRAPPED_INTERNALVAR (name)`
: This is an ugly hook to allow the specification of special actions that should occur as a side-effect of setting the value of a variable internal to GDB. Currently only used by the h8500. Note that this could be either a host or target conditional.

`NEED_TEXT_START_END`
: Define this if GDB should determine the start and end addresses of the text section. (Seems dubious.)

`NO_HIF_SUPPORT`
: (Specific to the a29k.)

`SOFTWARE_SINGLE_STEP_P`
: Define this as 1 if the target does not have a hardware single-step mechanism. The macro `SOFTWARE_SINGLE_STEP` must also be defined.

`SOFTWARE_SINGLE_STEP(signal,insert_breapoints_p)`
A function that inserts or removes (dependant on *insert_breapoints_p*) breakpoints at each possible destinations of the next instruction. See `sparc-tdep.c` and `rs6000-tdep.c` for examples.

`PCC_SOL_BROKEN`
(Used only in the Convex target.)

`PC_IN_CALL_DUMMY`
inferior.h

`PC_LOAD_SEGMENT`
If defined, print information about the load segment for the program counter. (Defined only for the RS/6000.)

`PC_REGNUM`
If the program counter is kept in a register, then define this macro to be the number (greater than or equal to zero) of that register.

This should only need to be defined if `TARGET_READ_PC` and `TARGET_WRITE_PC` are not defined.

`NPC_REGNUM`
The number of the "next program counter" register, if defined.

`NNPC_REGNUM`
The number of the "next next program counter" register, if defined. Currently, this is only defined for the Motorola 88K.

`PRINT_REGISTER_HOOK (regno)`
If defined, this must be a function that prints the contents of the given register to standard output.

`PRINT_TYPELESS_INTEGER`
This is an obscure substitute for `print_longest` that seems to have been defined for the Convex target.

`PROCESS_LINENUMBER_HOOK`
A hook defined for XCOFF reading.

`PROLOGUE_FIRSTLINE_OVERLAP`
(Only used in unsupported Convex configuration.)

`PS_REGNUM`
If defined, this is the number of the processor status register. (This definition is only used in generic code when parsing "$ps".)

`POP_FRAME`
Used in '`call_function_by_hand`' to remove an artificial stack frame.

`PUSH_ARGUMENTS (nargs, args, sp, struct_return, struct_addr)`
Define this to push arguments onto the stack for inferior function call. Return the updated stack pointer value.

`PUSH_DUMMY_FRAME`
Used in '`call_function_by_hand`' to create an artificial stack frame.

`REGISTER_BYTES`
: The total amount of space needed to store GDB's copy of the machine's register state.

`REGISTER_NAME(i)`
: Return the name of register *i* as a string. May return *NULL* or *NUL* to indicate that register *i* is not valid.

`REGISTER_NAMES`
: Deprecated in favor of *REGISTER_NAME*.

`REG_STRUCT_HAS_ADDR (gcc_p, type)`
: Define this to return 1 if the given type will be passed by pointer rather than directly.

`SAVE_DUMMY_FRAME_TOS (sp)`
: Used in '`call_function_by_hand`' to notify the target dependent code of the top-of-stack value that will be passed to the the inferior code. This is the value of the *SP* after both the dummy frame and space for parameters/results have been allocated on the stack.

`SDB_REG_TO_REGNUM`
: Define this to convert sdb register numbers into GDB regnums. If not defined, no conversion will be done.

`SHIFT_INST_REGS`
: (Only used for m88k targets.)

`SKIP_PROLOGUE (pc)`
: A C expression that returns the address of the "real" code beyond the function entry prologue found at *pc*.

`SKIP_PROLOGUE_FRAMELESS_P`
: A C expression that should behave similarly, but that can stop as soon as the function is known to have a frame. If not defined, `SKIP_PROLOGUE` will be used instead.

`SKIP_TRAMPOLINE_CODE (pc)`
: If the target machine has trampoline code that sits between callers and the functions being called, then define this macro to return a new PC that is at the start of the real function.

`SP_REGNUM`
: If the stack-pointer is kept in a register, then define this macro to be the number (greater than or equal to zero) of that register.

: This should only need to be defined if `TARGET_WRITE_SP` and `TARGET_WRITE_SP` are not defined.

`STAB_REG_TO_REGNUM`
: Define this to convert stab register numbers (as gotten from 'r' declarations) into GDB regnums. If not defined, no conversion will be done.

`STACK_ALIGN (addr)`
: Define this to adjust the address to the alignment required for the processor's stack.

`STEP_SKIPS_DELAY (addr)`
: Define this to return true if the address is of an instruction with a delay slot. If a breakpoint has been placed in the instruction's delay slot, GDB will single-step over that instruction before resuming normally. Currently only defined for the Mips.

`STORE_RETURN_VALUE (type, valbuf)`
: A C expression that stores a function return value of type *type*, where *valbuf* is the address of the value to be stored.

`SUN_FIXED_LBRAC_BUG`
: (Used only for Sun-3 and Sun-4 targets.)

`SYMBOL_RELOADING_DEFAULT`
: The default value of the 'symbol-reloading' variable. (Never defined in current sources.)

`TARGET_BYTE_ORDER_DEFAULT`
: The ordering of bytes in the target. This must be either `BIG_ENDIAN` or `LITTLE_ENDIAN`. This macro replaces *TARGET_BYTE_ORDER* which is deprecated.

`TARGET_BYTE_ORDER_SELECTABLE_P`
: Non-zero if the target has both `BIG_ENDIAN` and `LITTLE_ENDIAN` variants. This macro replaces *TARGET_BYTE_ORDER_SELECTABLE* which is deprecated.

`TARGET_CHAR_BIT`
: Number of bits in a char; defaults to 8.

`TARGET_COMPLEX_BIT`
: Number of bits in a complex number; defaults to `2 * TARGET_FLOAT_BIT`.

  At present this macro is not used.

`TARGET_DOUBLE_BIT`
: Number of bits in a double float; defaults to `8 * TARGET_CHAR_BIT`.

`TARGET_DOUBLE_COMPLEX_BIT`
: Number of bits in a double complex; defaults to `2 * TARGET_DOUBLE_BIT`.

  At present this macro is not used.

`TARGET_FLOAT_BIT`
: Number of bits in a float; defaults to `4 * TARGET_CHAR_BIT`.

`TARGET_INT_BIT`
: Number of bits in an integer; defaults to `4 * TARGET_CHAR_BIT`.

`TARGET_LONG_BIT`
: Number of bits in a long integer; defaults to `4 * TARGET_CHAR_BIT`.

`TARGET_LONG_DOUBLE_BIT`
: Number of bits in a long double float; defaults to `2 * TARGET_DOUBLE_BIT`.

`TARGET_LONG_LONG_BIT`
: Number of bits in a long long integer; defaults to `2 * TARGET_LONG_BIT`.

`TARGET_PTR_BIT`
: Number of bits in a pointer; defaults to `TARGET_INT_BIT`.

`TARGET_SHORT_BIT`
: Number of bits in a short integer; defaults to `2 * TARGET_CHAR_BIT`.

`TARGET_READ_PC`

`TARGET_WRITE_PC (val, pid)`

`TARGET_READ_SP`

`TARGET_WRITE_SP`

`TARGET_READ_FP`

`TARGET_WRITE_FP`
: These change the behavior of `read_pc`, `write_pc`, `read_sp`, `write_sp`, `read_fp` and `write_fp`. For most targets, these may be left undefined. GDB will call the read and write register functions with the relevant `_REGNUM` argument.

  These macros are useful when a target keeps one of these registers in a hard to get at place; for example, part in a segment register and part in an ordinary register.

`TARGET_VIRTUAL_FRAME_POINTER(pc,regp,offsetp)`
: Returns a `(register, offset)` pair representing the virtual frame pointer in use at the code address `"pc"`. If virtual frame pointers are not used, a default definition simply returns `FP_REGNUM`, with an offset of zero.

`USE_STRUCT_CONVENTION (gcc_p, type)`
: If defined, this must be an expression that is nonzero if a value of the given *type* being returned from a function must have space allocated for it on the stack. *gcc_p* is true if the function being considered is known to have been compiled by GCC; this is helpful for systems where GCC is known to use different calling convention than other compilers.

`VARIABLES_INSIDE_BLOCK (desc, gcc_p)`
: For dbx-style debugging information, if the compiler puts variable declarations inside LBRAC/RBRAC blocks, this should be defined to be nonzero. *desc* is the value of `n_desc` from the `N_RBRAC` symbol, and *gcc_p* is true if GDB has noticed the presence of either the `GCC_COMPILED_SYMBOL` or the `GCC2_COMPILED_SYMBOL`. By default, this is 0.

`OS9K_VARIABLES_INSIDE_BLOCK (desc, gcc_p)`
: Similarly, for OS/9000. Defaults to 1.

Motorola M68K target conditionals.

`BPT_VECTOR`
: Define this to be the 4-bit location of the breakpoint trap vector. If not defined, it will default to `0xf`.

`REMOTE_BPT_VECTOR`
: Defaults to `1`.

9.6 Adding a New Target

The following files define a target to GDB:

'`gdb/config/arch/ttt.mt`'
: Contains a Makefile fragment specific to this target. Specifies what object files are needed for target *ttt*, by defining '`TDEPFILES=...`'. Also specifies the header file which describes *ttt*, by defining '`TM_FILE= tm-ttt.h`'. You can also define '`TM_CFLAGS`', '`TM_CLIBS`', '`TM_CDEPS`', but these are now deprecated and may go away in future versions of GDB.

'`gdb/config/arch/tm-ttt.h`'
: ('`tm.h`' is a link to this file, created by configure). Contains macro definitions about the target machine's registers, stack frame format and instructions.

'`gdb/ttt-tdep.c`'
: Contains any miscellaneous code required for this target machine. On some machines it doesn't exist at all. Sometimes the macros in '`tm-ttt.h`' become very complicated, so they are implemented as functions here instead, and the macro is simply defined to call the function. This is vastly preferable, since it is easier to understand and debug.

'`gdb/config/arch/tm-arch.h`'
: This often exists to describe the basic layout of the target machine's processor chip (registers, stack, etc). If used, it is included by '`tm-ttt.h`'. It can be shared among many targets that use the same processor.

'`gdb/arch-tdep.c`'
: Similarly, there are often common subroutines that are shared by all target machines that use this particular architecture.

If you are adding a new operating system for an existing CPU chip, add a '`config/tm-os.h`' file that describes the operating system facilities that are unusual (extra symbol table info; the breakpoint instruction needed; etc). Then write a '`arch/tm-os.h`' that just `#includes` '`tm-arch.h`' and '`config/tm-os.h`'.

10 Target Vector Definition

The target vector defines the interface between GDB's abstract handling of target systems, and the nitty-gritty code that actually exercises control over a process or a serial port. GDB includes some 30-40 different target vectors; however, each configuration of GDB includes only a few of them.

10.1 File Targets

Both executables and core files have target vectors.

10.2 Standard Protocol and Remote Stubs

GDB's file '`remote.c`' talks a serial protocol to code that runs in the target system. GDB provides several sample "stubs" that can be integrated into target programs or operating systems for this purpose; they are named '`*-stub.c`'.

The GDB user's manual describes how to put such a stub into your target code. What follows is a discussion of integrating the SPARC stub into a complicated operating system (rather than a simple program), by Stu Grossman, the author of this stub.

The trap handling code in the stub assumes the following upon entry to trap_low:

1. %l1 and %l2 contain pc and npc respectively at the time of the trap
2. traps are disabled
3. you are in the correct trap window

As long as your trap handler can guarantee those conditions, then there is no reason why you shouldn't be able to 'share' traps with the stub. The stub has no requirement that it be jumped to directly from the hardware trap vector. That is why it calls `exceptionHandler()`, which is provided by the external environment. For instance, this could setup the hardware traps to actually execute code which calls the stub first, and then transfers to its own trap handler.

For the most point, there probably won't be much of an issue with 'sharing' traps, as the traps we use are usually not used by the kernel, and often indicate unrecoverable error conditions. Anyway, this is all controlled by a table, and is trivial to modify. The most important trap for us is for `ta 1`. Without that, we can't single step or do breakpoints. Everything else is unnecessary for the proper operation of the debugger/stub.

From reading the stub, it's probably not obvious how breakpoints work. They are simply done by deposit/examine operations from GDB.

10.3 ROM Monitor Interface

10.4 Custom Protocols

10.5 Transport Layer

10.6 Builtin Simulator

11 Native Debugging

Several files control GDB's configuration for native support:

'`gdb/config/`*arch*`/`*xyz*`.mh`'

Specifies Makefile fragments needed when hosting *or native* on machine *xyz*. In particular, this lists the required native-dependent object files, by defining '`NATDEPFILES=...`'. Also specifies the header file which describes native support on *xyz*, by defining '`NAT_FILE= nm-`*xyz*`.h`'. You can also define '`NAT_CFLAGS`', '`NAT_ADD_FILES`', '`NAT_CLIBS`', '`NAT_CDEPS`', etc.; see '`Makefile.in`'.

'`gdb/config/`*arch*`/nm-`*xyz*`.h`'
: ('`nm.h`' is a link to this file, created by configure). Contains C macro definitions describing the native system environment, such as child process control and core file support.

'`gdb/`*xyz*`-nat.c`'
: Contains any miscellaneous C code required for this native support of this machine. On some machines it doesn't exist at all.

There are some "generic" versions of routines that can be used by various systems. These can be customized in various ways by macros defined in your '`nm-`*xyz*`.h`' file. If these routines work for the *xyz* host, you can just include the generic file's name (with '`.o`', not '`.c`') in `NATDEPFILES`.

Otherwise, if your machine needs custom support routines, you will need to write routines that perform the same functions as the generic file. Put them into *xyz*`-nat.c`, and put *xyz*`-nat.o` into `NATDEPFILES`.

'`inftarg.c`'
: This contains the *target_ops vector* that supports Unix child processes on systems which use ptrace and wait to control the child.

'`procfs.c`'
: This contains the *target_ops vector* that supports Unix child processes on systems which use /proc to control the child.

'`fork-child.c`'
: This does the low-level grunge that uses Unix system calls to do a "fork and exec" to start up a child process.

'`infptrace.c`'
: This is the low level interface to inferior processes for systems using the Unix `ptrace` call in a vanilla way.

11.1 Native core file Support

'`core-aout.c::fetch_core_registers()`'
: Support for reading registers out of a core file. This routine calls `register_addr()`, see below. Now that BFD is used to read core files, virtually all machines should use `core-aout.c`, and should just provide `fetch_core_registers` in *xyz*`-nat.c` (or `REGISTER_U_ADDR` in `nm-`*xyz*`.h`).

'`core-aout.c::register_addr()`'
: If your `nm-`*xyz*`.h` file defines the macro `REGISTER_U_ADDR(addr, blockend, regno)`, it should be defined to set `addr` to the offset within the '`user`' struct of GDB register number `regno`. `blockend` is the offset within the "upage" of `u.u_ar0`. If `REGISTER_U_ADDR` is defined, '`core-aout.c`' will define the `register_addr()` function and use the macro in it. If you do not define `REGISTER_U_ADDR`, but you are using the standard `fetch_core_registers()`, you will need to define your own version of `register_addr()`, put it into your *xyz*`-nat.c` file, and be sure *xyz*`-nat.o` is in the `NATDEPFILES` list. If you have your own `fetch_core_registers()`, you may not need a separate `register_addr()`. Many

custom `fetch_core_registers()` implementations simply locate the registers themselves.

When making GDB run native on a new operating system, to make it possible to debug core files, you will need to either write specific code for parsing your OS's core files, or customize '`bfd/trad-core.c`'. First, use whatever `#include` files your machine uses to define the struct of registers that is accessible (possibly in the u-area) in a core file (rather than '`machine/reg.h`'), and an include file that defines whatever header exists on a core file (e.g. the u-area or a '`struct core`'). Then modify `trad_unix_core_file_p()` to use these values to set up the section information for the data segment, stack segment, any other segments in the core file (perhaps shared library contents or control information), "registers" segment, and if there are two discontiguous sets of registers (e.g. integer and float), the "reg2" segment. This section information basically delimits areas in the core file in a standard way, which the section-reading routines in BFD know how to seek around in.

Then back in GDB, you need a matching routine called `fetch_core_registers()`. If you can use the generic one, it's in '`core-aout.c`'; if not, it's in your '*xyz*`-nat.c`' file. It will be passed a char pointer to the entire "registers" segment, its length, and a zero; or a char pointer to the entire "regs2" segment, its length, and a 2. The routine should suck out the supplied register values and install them into GDB's "registers" array.

If your system uses '`/proc`' to control processes, and uses ELF format core files, then you may be able to use the same routines for reading the registers out of processes and out of core files.

11.2 ptrace

11.3 /proc

11.4 win32

11.5 shared libraries

11.6 Native Conditionals

When GDB is configured and compiled, various macros are defined or left undefined, to control compilation when the host and target systems are the same. These macros should be defined (or left undefined) in '`nm-`*system*`.h`'.

`ATTACH_DETACH`
: If defined, then GDB will include support for the `attach` and `detach` commands.

`CHILD_PREPARE_TO_STORE`
: If the machine stores all registers at once in the child process, then define this to ensure that all values are correct. This usually entails a read from the child.

  [Note that this is incorrectly defined in '`xm-`*system*`.h`' files currently.]

`FETCH_INFERIOR_REGISTERS`
Define this if the native-dependent code will provide its own routines `fetch_inferior_registers` and `store_inferior_registers` in '*HOST*-`nat.c`'. If this symbol is *not* defined, and '`infptrace.c`' is included in this configuration, the default routines in '`infptrace.c`' are used for these functions.

`FILES_INFO_HOOK`
(Only defined for Convex.)

`FP0_REGNUM`
This macro is normally defined to be the number of the first floating point register, if the machine has such registers. As such, it would appear only in target-specific code. However, /proc support uses this to decide whether floats are in use on this target.

`GET_LONGJMP_TARGET`
For most machines, this is a target-dependent parameter. On the DECstation and the Iris, this is a native-dependent parameter, since <setjmp.h> is needed to define it.

This macro determines the target PC address that longjmp() will jump to, assuming that we have just stopped at a longjmp breakpoint. It takes a CORE_ADDR * as argument, and stores the target PC value through this pointer. It examines the current state of the machine as needed.

`KERNEL_U_ADDR`
Define this to the address of the `u` structure (the "user struct", also known as the "u-page") in kernel virtual memory. GDB needs to know this so that it can subtract this address from absolute addresses in the upage, that are obtained via ptrace or from core files. On systems that don't need this value, set it to zero.

`KERNEL_U_ADDR_BSD`
Define this to cause GDB to determine the address of `u` at runtime, by using Berkeley-style `nlist` on the kernel's image in the root directory.

`KERNEL_U_ADDR_HPUX`
Define this to cause GDB to determine the address of `u` at runtime, by using HP-style `nlist` on the kernel's image in the root directory.

`ONE_PROCESS_WRITETEXT`
Define this to be able to, when a breakpoint insertion fails, warn the user that another process may be running with the same executable.

`PROC_NAME_FMT`
Defines the format for the name of a '`/proc`' device. Should be defined in '`nm.h`' *only* in order to override the default definition in '`procfs.c`'.

`PTRACE_FP_BUG`
mach386-xdep.c

`PTRACE_ARG3_TYPE`
The type of the third argument to the `ptrace` system call, if it exists and is different from `int`.

`REGISTER_U_ADDR`
: Defines the offset of the registers in the "u area".

`SHELL_COMMAND_CONCAT`
: If defined, is a string to prefix on the shell command used to start the inferior.

`SHELL_FILE`
: If defined, this is the name of the shell to use to run the inferior. Defaults to `"/bin/sh"`.

`SOLIB_ADD (filename, from_tty, targ)`
: Define this to expand into an expression that will cause the symbols in *filename* to be added to GDB's symbol table.

`SOLIB_CREATE_INFERIOR_HOOK`
: Define this to expand into any shared-library-relocation code that you want to be run just after the child process has been forked.

`START_INFERIOR_TRAPS_EXPECTED`
: When starting an inferior, GDB normally expects to trap twice; once when the shell execs, and once when the program itself execs. If the actual number of traps is something other than 2, then define this macro to expand into the number expected.

`SVR4_SHARED_LIBS`
: Define this to indicate that SVR4-style shared libraries are in use.

`USE_PROC_FS`
: This determines whether small routines in '`*-tdep.c`', which translate register values between GDB's internal representation and the /proc representation, are compiled.

`U_REGS_OFFSET`
: This is the offset of the registers in the upage. It need only be defined if the generic ptrace register access routines in '`infptrace.c`' are being used (that is, '`infptrace.c`' is configured in, and `FETCH_INFERIOR_REGISTERS` is not defined). If the default value from '`infptrace.c`' is good enough, leave it undefined.

: The default value means that u.u_ar0 *points to* the location of the registers. I'm guessing that `#define U_REGS_OFFSET 0` means that u.u_ar0 *is* the location of the registers.

`CLEAR_SOLIB`
: objfiles.c

`DEBUG_PTRACE`
: Define this to debug ptrace calls.

12 Support Libraries

12.1 BFD

BFD provides support for GDB in several ways:

identifying executable and core files
: BFD will identify a variety of file types, including a.out, coff, and several variants thereof, as well as several kinds of core files.

access to sections of files
: BFD parses the file headers to determine the names, virtual addresses, sizes, and file locations of all the various named sections in files (such as the text section or the data section). GDB simply calls BFD to read or write section X at byte offset Y for length Z.

specialized core file support
: BFD provides routines to determine the failing command name stored in a core file, the signal with which the program failed, and whether a core file matches (i.e. could be a core dump of) a particular executable file.

locating the symbol information
: GDB uses an internal interface of BFD to determine where to find the symbol information in an executable file or symbol-file. GDB itself handles the reading of symbols, since BFD does not "understand" debug symbols, but GDB uses BFD's cached information to find the symbols, string table, etc.

12.2 opcodes

The opcodes library provides GDB's disassembler. (It's a separate library because it's also used in binutils, for '`objdump`').

12.3 readline

12.4 mmalloc

12.5 libiberty

12.6 gnu-regex

Regex conditionals.

`C_ALLOCA`

`NFAILURES`

`RE_NREGS`

`SIGN_EXTEND_CHAR`

`SWITCH_ENUM_BUG`

`SYNTAX_TABLE`

`Sword`

`sparc`

12.7 include

13 Coding

This chapter covers topics that are lower-level than the major algorithms of GDB.

13.1 Cleanups

Cleanups are a structured way to deal with things that need to be done later. When your code does something (like `malloc` some memory, or open a file) that needs to be undone later (e.g. free the memory or close the file), it can make a cleanup. The cleanup will be done at some future point: when the command is finished, when an error occurs, or when your code decides it's time to do cleanups.

You can also discard cleanups, that is, throw them away without doing what they say. This is only done if you ask that it be done.

Syntax:

`struct cleanup *`*old_chain*`;`
: Declare a variable which will hold a cleanup chain handle.

old_chain `= make_cleanup (`*function*`,` *arg*`);`
: Make a cleanup which will cause *function* to be called with *arg* (a `char *`) later. The result, *old_chain*, is a handle that can be passed to `do_cleanups` or `discard_cleanups` later. Unless you are going to call `do_cleanups` or `discard_cleanups` yourself, you can ignore the result from `make_cleanup`.

`do_cleanups (`*old_chain*`);`
: Perform all cleanups done since `make_cleanup` returned *old_chain*. E.g.:

```
make_cleanup (a, 0);
old = make_cleanup (b, 0);
do_cleanups (old);
```

will call `b()` but will not call `a()`. The cleanup that calls `a()` will remain in the cleanup chain, and will be done later unless otherwise discarded.

`discard_cleanups (`*old_chain*`);`
: Same as `do_cleanups` except that it just removes the cleanups from the chain and does not call the specified functions.

Some functions, e.g. `fputs_filtered()` or `error()`, specify that they "should not be called when cleanups are not in place". This means that any actions you need to reverse in the case of an error or interruption must be on the cleanup chain before you call these functions, since they might never return to your code (they '`longjmp`' instead).

13.2 Wrapping Output Lines

Output that goes through `printf_filtered` or `fputs_filtered` or `fputs_demangled` needs only to have calls to `wrap_here` added in places that would be good breaking points. The utility routines will take care of actually wrapping if the line width is exceeded.

The argument to `wrap_here` is an indentation string which is printed *only* if the line breaks there. This argument is saved away and used later. It must remain valid until the next call to `wrap_here` or until a newline has been printed through the `*_filtered` functions. Don't pass in a local variable and then return!

It is usually best to call `wrap_here()` after printing a comma or space. If you call it before printing a space, make sure that your indentation properly accounts for the leading space that will print if the line wraps there.

Any function or set of functions that produce filtered output must finish by printing a newline, to flush the wrap buffer, before switching to unfiltered ("`printf`") output. Symbol reading routines that print warnings are a good example.

13.3 GDB Coding Standards

GDB follows the GNU coding standards, as described in '`etc/standards.texi`'. This file is also available for anonymous FTP from GNU archive sites. GDB takes a strict interpretation of the standard; in general, when the GNU standard recommends a practice but does not require it, GDB requires it.

GDB follows an additional set of coding standards specific to GDB, as described in the following sections.

You can configure with '`--enable-build-warnings`' to get GCC to check on a number of these rules. GDB sources ought not to engender any complaints, unless they are caused by bogus host systems. (The exact set of enabled warnings is currently '`-Wall -Wpointer-arith -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations`'.

13.3.1 Formatting

The standard GNU recommendations for formatting must be followed strictly.

Note that while in a definition, the function's name must be in column zero; in a function declaration, the name must be on the same line as the return type.

In addition, there must be a space between a function or macro name and the opening parenthesis of its argument list (except for macro definitions, as required by C). There must not be a space after an open paren/bracket or before a close paren/bracket.

While additional whitespace is generally helpful for reading, do not use more than one blank line to separate blocks, and avoid adding whitespace after the end of a program line (as of 1/99, some 600 lines had whitespace after the semicolon). Excess whitespace causes difficulties for diff and patch.

13.3.2 Comments

The standard GNU requirements on comments must be followed strictly.

Block comments must appear in the following form, with no '/*'- or '*/'-only lines, and no leading '*':

```
/* Wait for control to return from inferior to debugger.  If inferior
   gets a signal, we may decide to start it up again instead of
   returning.  That is why there is a loop in this function.  When
   this function actually returns it means the inferior should be left
   stopped and GDB should read more commands.  */
```

(Note that this format is encouraged by Emacs; tabbing for a multi-line comment works correctly, and M-Q fills the block consistently.)

Put a blank line between the block comments preceding function or variable definitions, and the definition itself.

In general, put function-body comments on lines by themselves, rather than trying to fit them into the 20 characters left at the end of a line, since either the comment or the code will inevitably get longer than will fit, and then somebody will have to move it anyhow.

13.3.3 C Usage

Code must not depend on the sizes of C data types, the format of the host's floating point numbers, the alignment of anything, or the order of evaluation of expressions.

Use functions freely. There are only a handful of compute-bound areas in GDB that might be affected by the overhead of a function call, mainly in symbol reading. Most of GDB's performance is limited by the target interface (whether serial line or system call).

However, use functions with moderation. A thousand one-line functions are just as hard to understand as a single thousand-line function.

13.3.4 Function Prototypes

Prototypes must be used to *declare* functions but never to *define* them. Prototypes for GDB functions must include both the argument type and name, with the name matching that used in the actual function definition.

For the sake of compatibility with pre-ANSI compilers, define prototypes with the `PARAMS` macro:

```
extern int memory_remove_breakpoint PARAMS ((CORE_ADDR addr,
                                             char *contents_cache));
```

Note the double parentheses around the parameter types. This allows an arbitrary number of parameters to be described, without freaking out the C preprocessor. When the function has no parameters, it should be described like:

```
extern void noprocess PARAMS ((void));
```

The `PARAMS` macro expands to its argument in ANSI C, or to a simple `()` in traditional C.

All external functions should have a `PARAMS` declaration in a header file that callers include, except for `_initialize_*` functions, which must be external so that '`init.c`' construction works, but shouldn't be visible to random source files.

All static functions must be declared in a block near the top of the source file.

13.3.5 Clean Design

In addition to getting the syntax right, there's the little question of semantics. Some things are done in certain ways in GDB because long experience has shown that the more obvious ways caused various kinds of trouble.

You can't assume the byte order of anything that comes from a target (including *values*, object files, and instructions). Such things must be byte-swapped using `SWAP_TARGET_AND_HOST` in GDB, or one of the swap routines defined in '`bfd.h`', such as `bfd_get_32`.

You can't assume that you know what interface is being used to talk to the target system. All references to the target must go through the current `target_ops` vector.

You can't assume that the host and target machines are the same machine (except in the "native" support modules). In particular, you can't assume that the target machine's header files will be available on the host machine. Target code must bring along its own header files – written from scratch or explicitly donated by their owner, to avoid copyright problems.

Insertion of new `#ifdef`'s will be frowned upon. It's much better to write the code portably than to conditionalize it for various systems.

New `#ifdef`'s which test for specific compilers or manufacturers or operating systems are unacceptable. All `#ifdef`'s should test for features. The information about which configurations contain which features should be segregated into the configuration files. Experience has proven far too often that a feature unique to one particular system often creeps into other systems; and that a conditional based on some predefined macro for your current system will become worthless over time, as new versions of your system come out that behave differently with regard to this feature.

Adding code that handles specific architectures, operating systems, target interfaces, or hosts, is not acceptable in generic code. If a hook is needed at that point, invent a generic hook and define it for your configuration, with something like:

```
#ifdef WRANGLE_SIGNALS
   WRANGLE_SIGNALS (signo);
#endif
```

In your host, target, or native configuration file, as appropriate, define `WRANGLE_SIGNALS` to do the machine-dependent thing. Take a bit of care in defining the hook, so that it can be used by other ports in the future, if they need a hook in the same place.

If the hook is not defined, the code should do whatever "most" machines want. Using `#ifdef`, as above, is the preferred way to do this, but sometimes that gets convoluted, in which case use

```
#ifndef SPECIAL_FOO_HANDLING
#define SPECIAL_FOO_HANDLING(pc, sp) (0)
#endif
```

where the macro is used or in an appropriate header file.

Whether to include a *small* hook, a hook around the exact pieces of code which are system-dependent, or whether to replace a whole function with a hook depends on the case. A good example of this dilemma can be found in `get_saved_register`. All machines that GDB 2.8 ran on just needed the `FRAME_FIND_SAVED_REGS` hook to find the saved registers. Then the SPARC and Pyramid came along, and `HAVE_REGISTER_WINDOWS` and `REGISTER_IN_WINDOW_P` were introduced. Then the 29k and 88k required the `GET_SAVED_REGISTER` hook. The first three are examples of small hooks; the latter replaces a whole function. In this specific case, it is useful to have both kinds; it would be a bad idea to replace all the uses of the small hooks with `GET_SAVED_REGISTER`, since that would result in much duplicated code. Other times, duplicating a few lines of code here or there is much cleaner than introducing a large number of small hooks.

Another way to generalize GDB along a particular interface is with an attribute struct. For example, GDB has been generalized to handle multiple kinds of remote interfaces – not by #ifdef's everywhere, but by defining the "target_ops" structure and having a current target (as well as a stack of targets below it, for memory references). Whenever something needs to be done that depends on which remote interface we are using, a flag in the current target_ops structure is tested (e.g. 'target_has_stack'), or a function is called through a pointer in the current target_ops structure. In this way, when a new remote interface is added, only one module needs to be touched – the one that actually implements the new remote interface. Other examples of attribute-structs are BFD access to multiple kinds of object file formats, or GDB's access to multiple source languages.

Please avoid duplicating code. For example, in GDB 3.x all the code interfacing between `ptrace` and the rest of GDB was duplicated in '`*-dep.c`', and so changing something was very painful. In GDB 4.x, these have all been consolidated into '`infptrace.c`'. '`infptrace.c`' can deal with variations between systems the same way any system-independent file would (hooks, #if defined, etc.), and machines which are radically different don't need to use infptrace.c at all.

Don't put debugging printfs in the code.

14 Porting GDB

Most of the work in making GDB compile on a new machine is in specifying the configuration of the machine. This is done in a dizzying variety of header files and configuration scripts, which we hope to make more sensible soon. Let's say your new host is called an *xyz* (e.g. '`sun4`'), and its full three-part configuration name is *arch-xvend-xos* (e.g. '`sparc-sun-sunos4`'). In particular:

In the top level directory, edit '`config.sub`' and add *arch*, *xvend*, and *xos* to the lists of supported architectures, vendors, and operating systems near the bottom of the file. Also, add *xyz* as an alias that maps to *arch-xvend-xos*. You can test your changes by running

```
./config.sub xyz
```

and

```
./config.sub arch-xvend-xos
```

which should both respond with *arch-xvend-xos* and no error messages.

You need to port BFD, if that hasn't been done already. Porting BFD is beyond the scope of this manual.

To configure GDB itself, edit '`gdb/configure.host`' to recognize your system and set `gdb_host` to *xyz*, and (unless your desired target is already available) also edit '`gdb/configure.tgt`', setting `gdb_target` to something appropriate (for instance, *xyz*).

Finally, you'll need to specify and define GDB's host-, native-, and target-dependent '`.h`' and '`.c`' files used for your configuration.

14.1 Configuring GDB for Release

From the top level directory (containing '`gdb`', '`bfd`', '`libiberty`', and so on):

```
make -f Makefile.in gdb.tar.gz
```

This will properly configure, clean, rebuild any files that are distributed pre-built (e.g. '`c-exp.tab.c`' or '`refcard.ps`'), and will then make a tarfile. (If the top level directory has already been configured, you can just do `make gdb.tar.gz` instead.)

This procedure requires:

- symbolic links
- `makeinfo` (texinfo2 level)
- TEX
- `dvips`
- `yacc` or `bison`

. . . and the usual slew of utilities (`sed`, `tar`, etc.).

TEMPORARY RELEASE PROCEDURE FOR DOCUMENTATION

'`gdb.texinfo`' is currently marked up using the texinfo-2 macros, which are not yet a default for anything (but we have to start using them sometime).

For making paper, the only thing this implies is the right generation of '`texinfo.tex`' needs to be included in the distribution.

For making info files, however, rather than duplicating the texinfo2 distribution, generate '`gdb-all.texinfo`' locally, and include the files '`gdb.info*`' in the distribution. Note the plural; `makeinfo` will split the document into one overall file and five or so included files.

15 Testsuite

The testsuite is an important component of the GDB package. While it is always worthwhile to encourage user testing, in practice this is rarely sufficient; users typically use only a small subset of the available commands, and it has proven all too common for a change to cause a significant regression that went unnoticed for some time.

The GDB testsuite uses the DejaGNU testing framework. DejaGNU is built using tcl and expect. The tests themselves are calls to various tcl procs; the framework runs all the procs and summarizes the passes and fails.

15.1 Using the Testsuite

To run the testsuite, simply go to the GDB object directory (or to the testsuite's objdir) and type `make check`. This just sets up some environment variables and invokes DejaGNU's `runtest` script. While the testsuite is running, you'll get mentions of which test file is in use, and a mention of any unexpected passes or fails. When the testsuite is finished, you'll get a summary that looks like this:

```
                    === gdb Summary ===

# of expected passes            6016
# of unexpected failures        58
# of unexpected successes       5
# of expected failures          183
# of unresolved testcases       3
# of untested testcases         5
```

The ideal test run consists of expected passes only; however, reality conspires to keep us from this ideal. Unexpected failures indicate real problems, whether in GDB or in the testsuite. Expected failures are still failures, but ones which have been decided are too hard to deal with at the time; for instance, a test case might work everywhere except on AIX, and there is no prospect of the AIX case being fixed in the near future. Expected failures should not be added lightly, since you may be masking serious bugs in GDB. Unexpected successes are expected fails that are passing for some reason, while unresolved and untested cases often indicate some minor catastrophe, such as the compiler being unable to deal with a test program.

When making any significant change to GDB, you should run the testsuite before and after the change, to confirm that there are no regressions. Note that truly complete testing would require that you run the testsuite with all supported configurations and a variety of compilers; however this is more than really necessary. In many cases testing with a single configuration is sufficient. Other useful options are to test one big-endian (Sparc) and one little-endian (x86) host, a cross config with a builtin simulator (powerpc-eabi, mips-elf), or a 64-bit host (Alpha).

If you add new functionality to GDB, please consider adding tests for it as well; this way future GDB hackers can detect and fix their changes that break the functionality you added. Similarly, if you fix a bug that was not previously reported as a test failure, please add a test case for it. Some cases are extremely difficult to test, such as code that handles host OS failures or bugs in particular versions of compilers, and it's OK not to try to write tests for all of those.

15.2 Testsuite Organization

The testsuite is entirely contained in '`gdb/testsuite`'. While the testsuite includes some makefiles and configury, these are very minimal, and used for little besides cleaning up, since the tests themselves handle the compilation of the programs that GDB will run. The file '`testsuite/lib/gdb.exp`' contains common utility procs useful for all GDB tests, while the directory '`testsuite/config`' contains configuration-specific files, typically used for special-purpose definitions of procs like `gdb_load` and `gdb_start`.

The tests themselves are to be found in '`testsuite/gdb.*`' and subdirectories of those. The names of the test files must always end with '`.exp`'. DejaGNU collects the test files by wildcarding in the test directories, so both subdirectories and individual files get chosen and run in alphabetical order.

The following table lists the main types of subdirectories and what they are for. Since DejaGNU finds test files no matter where they are located, and since each test file sets up its own compilation and execution environment, this organization is simply for convenience and intelligibility.

`gdb.base`
: This is the base testsuite. The tests in it should apply to all configurations of GDB (but generic native-only tests may live here). The test programs should be in the subset of C that is valid K&R, ANSI/ISO, and C++ (ifdefs are allowed if necessary, for instance for prototypes).

`gdb.`*lang*
: Language-specific tests for all languages besides C. Examples are '`gdb.c++`' and '`gdb.java`'.

`gdb.`*platform*
: Non-portable tests. The tests are specific to a specific configuration (host or target), such as HP-UX or eCos. Example is '`gdb.hp`', for HP-UX.

`gdb.`*compiler*
: Tests specific to a particular compiler. As of this writing (June 1999), there aren't currently any groups of tests in this category that couldn't just as sensibly be made platform-specific, but one could imagine a gdb.gcc, for tests of GDB's handling of GCC extensions.

`gdb.`*subsystem*
: Tests that exercise a specific GDB subsystem in more depth. For instance, '`gdb.disasm`' exercises various disassemblers, while '`gdb.stabs`' tests pathways through the stabs symbol reader.

15.3 Writing Tests

In many areas, the GDB tests are already quite comprehensive; you should be able to copy existing tests to handle new cases.

You should try to use `gdb_test` whenever possible, since it includes cases to handle all the unexpected errors that might happen. However, it doesn't cost anything to add new test procedures; for instance, '`gdb.base/exprs.exp`' defines a `test_expr` that calls `gdb_test` multiple times.

Only use `send_gdb` and `gdb_expect` when absolutely necessary, such as when GDB has several valid responses to a command.

The source language programs do *not* need to be in a consistent style. Since GDB is used to debug programs written in many different styles, it's worth having a mix of styles in the testsuite; for instance, some GDB bugs involving the display of source lines would never manifest themselves if the programs used GNU coding style uniformly.

16 Hints

Check the 'README' file, it often has useful information that does not appear anywhere else in the directory.

16.1 Getting Started

GDB is a large and complicated program, and if you first starting to work on it, it can be hard to know where to start. Fortunately, if you know how to go about it, there are ways to figure out what is going on.

This manual, the GDB Internals manual, has information which applies generally to many parts of GDB.

Information about particular functions or data structures are located in comments with those functions or data structures. If you run across a function or a global variable which does not have a comment correctly explaining what is does, this can be thought of as a bug in GDB; feel free to submit a bug report, with a suggested comment if you can figure out what the comment should say. If you find a comment which is actually wrong, be especially sure to report that.

Comments explaining the function of macros defined in host, target, or native dependent files can be in several places. Sometimes they are repeated every place the macro is defined. Sometimes they are where the macro is used. Sometimes there is a header file which supplies a default definition of the macro, and the comment is there. This manual also documents all the available macros.

Start with the header files. Once you some idea of how GDB's internal symbol tables are stored (see 'symtab.h', 'gdbtypes.h'), you will find it much easier to understand the code which uses and creates those symbol tables.

You may wish to process the information you are getting somehow, to enhance your understanding of it. Summarize it, translate it to another language, add some (perhaps trivial or non-useful) feature to GDB, use the code to predict what a test case would do and write the test case and verify your prediction, etc. If you are reading code and your eyes are starting to glaze over, this is a sign you need to use a more active approach.

Once you have a part of GDB to start with, you can find more specifically the part you are looking for by stepping through each function with the `next` command. Do not use `step` or you will quickly get distracted; when the function you are stepping through calls another function try only to get a big-picture understanding (perhaps using the comment at the beginning of the function being called) of what it does. This way you can identify which of the functions being called by the function you are stepping through is the one which you are interested in. You may need to examine the data structures generated at each stage, with reference to the comments in the header files explaining what the data structures are supposed to look like.

Of course, this same technique can be used if you are just reading the code, rather than actually stepping through it. The same general principle applies—when the code you are looking at calls something else, just try to understand generally what the code being called does, rather than worrying about all its details.

A good place to start when tracking down some particular area is with a command which invokes that feature. Suppose you want to know how single-stepping works. As a GDB user, you know that the `step` command invokes single-stepping. The command is invoked via command tables (see '`command.h`'); by convention the function which actually performs the command is formed by taking the name of the command and adding '`_command`', or in the case of an `info` subcommand, '`_info`'. For example, the `step` command invokes the `step_command` function and the `info display` command invokes `display_info`. When this convention is not followed, you might have to use `grep` or *`M-x tags-search`* in emacs, or run GDB on itself and set a breakpoint in `execute_command`.

If all of the above fail, it may be appropriate to ask for information on `bug-gdb`. But *never* post a generic question like "I was wondering if anyone could give me some tips about understanding GDB"—if we had some magic secret we would put it in this manual. Suggestions for improving the manual are always welcome, of course.

16.2 Debugging GDB with itself

If GDB is limping on your machine, this is the preferred way to get it fully functional. Be warned that in some ancient Unix systems, like Ultrix 4.2, a program can't be running in one process while it is being debugged in another. Rather than typing the command `./gdb ./gdb`, which works on Suns and such, you can copy '`gdb`' to '`gdb2`' and then type `./gdb ./gdb2`.

When you run GDB in the GDB source directory, it will read a '`.gdbinit`' file that sets up some simple things to make debugging gdb easier. The `info` command, when executed without a subcommand in a GDB being debugged by gdb, will pop you back up to the top level gdb. See '`.gdbinit`' for details.

If you use emacs, you will probably want to do a `make TAGS` after you configure your distribution; this will put the machine dependent routines for your local machine where they will be accessed first by *`M-.`*

Also, make sure that you've either compiled GDB with your local cc, or have run `fixincludes` if you are compiling with gcc.

16.3 Submitting Patches

Thanks for thinking of offering your changes back to the community of GDB users. In general we like to get well designed enhancements. Thanks also for checking in advance about the best way to transfer the changes.

The GDB maintainers will only install "cleanly designed" patches. This manual summarizes what we believe to be clean design for GDB.

If the maintainers don't have time to put the patch in when it arrives, or if there is any question about a patch, it goes into a large queue with everyone else's patches and bug reports.

The legal issue is that to incorporate substantial changes requires a copyright assignment from you and/or your employer, granting ownership of the changes to the Free Software Foundation. You can get the standard documents for doing this by sending mail to `gnu@gnu.org` and asking for it. We recommend that people write in "All programs owned by the Free Software Foundation" as "NAME OF PROGRAM", so that changes in many programs (not just GDB, but GAS, Emacs, GCC, etc) can be contributed with only one piece of legalese pushed through the bureacracy and filed with the FSF. We can't start merging changes until this paperwork is received by the FSF (their rules, which we follow since we maintain it for them).

Technically, the easiest way to receive changes is to receive each feature as a small context diff or unidiff, suitable for "patch". Each message sent to me should include the changes to C code and header files for a single feature, plus ChangeLog entries for each directory where files were modified, and diffs for any changes needed to the manuals (gdb/doc/gdb.texinfo or gdb/doc/gdbint.texinfo). If there are a lot of changes for a single feature, they can be split down into multiple messages.

In this way, if we read and like the feature, we can add it to the sources with a single patch command, do some testing, and check it in. If you leave out the ChangeLog, we have to write one. If you leave out the doc, we have to puzzle out what needs documenting. Etc.

The reason to send each change in a separate message is that we will not install some of the changes. They'll be returned to you with questions or comments. If we're doing our job correctly, the message back to you will say what you have to fix in order to make the change acceptable. The reason to have separate messages for separate features is so that the acceptable changes can be installed while one or more changes are being reworked. If multiple features are sent in a single message, we tend to not put in the effort to sort out the acceptable changes from the unacceptable, so none of the features get installed until all are acceptable.

If this sounds painful or authoritarian, well, it is. But we get a lot of bug reports and a lot of patches, and many of them don't get installed because we don't have the time to finish the job that the bug reporter or the contributor could have done. Patches that arrive complete, working, and well designed, tend to get installed on the day they arrive. The others go into a queue and get installed as time permits, which, since the maintainers have many demands to meet, may not be for quite some time.

Please send patches directly to the GDB maintainers at `gdb-patches@sourceware.cygnus.com`.

16.4 Obsolete Conditionals

Fragments of old code in GDB sometimes reference or set the following configuration macros. They should not be used by new code, and old uses should be removed as those parts of the debugger are otherwise touched.

`STACK_END_ADDR`

This macro used to define where the end of the stack appeared, for use in interpreting core file formats that don't record this address in the core file itself. This information is now configured in BFD, and GDB gets the info portably from there. The values in GDB's configuration files should be moved into BFD configuration files (if needed there), and deleted from all of GDB's config files.

Any '*foo*`-xdep.c`' file that references STACK_END_ADDR is so old that it has never been converted to use BFD. Now that's old!

`PYRAMID_CONTROL_FRAME_DEBUGGING`

pyr-xdep.c

`PYRAMID_CORE`
pyr-xdep.c

`PYRAMID_PTRACE`
pyr-xdep.c

`REG_STACK_SEGMENT`
exec.c

Table of Contents