GDB Internals

A guide to the internals of the GNU debugger

John Gilmore
Cygnus Solutions
Second Edition:
Stan Shebs
Cygnus Solutions

Table of Contents

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*).

The 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 (shorthand for *program counter*) 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. (On some architectures, notably the 32-bit x86 platforms, GDB cannot always know whether there's enough hardware resources to insert all the hardware breakpoints and watchpoints. On those platforms, GDB prints an error message only when the program being debugged is continued.)

Software breakpoints require GDB to do somewhat more work. The basic theory is that GDB will replace a program instruction with 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 writable, so this technique won't work on programs in ROM. It can also distort the behavior of programs that examine themselves, although such a 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 output of 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-`*target*`.h`' file. Look in '`tm-sun4os4.h`' and '`sparc-tdep.c`' for examples of how to do this.

3.8 Watchpoints

Watchpoints are a special kind of breakpoints (see Chapter 3 [Algorithms], page 2) which break when data is accessed rather than when some instruction is executed. When you have data which changes without your knowing what code does that, watchpoints are the silver bullet to hunt down and kill such bugs.

Watchpoints can be either hardware-assisted or not; the latter type is known as "software watchpoints." GDB always uses hardware-assisted watchpoints if they are available, and falls back on software watchpoints otherwise. Typical situations where GDB will use software watchpoints are:

- The watched memory region is too large for the underlying hardware watchpoint support. For example, each x86 debug register can watch up to 4 bytes of memory, so trying to watch data structures whose size is more than 16 bytes will cause GDB to use software watchpoints.

- The value of the expression to be watched depends on data held in registers (as opposed to memory).

- Too many different watchpoints requested. (On some architectures, this situation is impossible to detect until the debugged program is resumed.) Note that x86 debug registers are used both for hardware breakpoints and for watchpoints, so setting too many hardware breakpoints might cause watchpoint insertion to fail.

- No hardware-assisted watchpoints provided by the target implementation.

Software watchpoints are very slow, since GDB needs to single-step the program being debugged and test the value of the watched expression(s) after each instruction. The rest of this section is mostly irrelevant for software watchpoints.

GDB uses several macros and primitives to support hardware watchpoints:

`TARGET_HAS_HARDWARE_WATCHPOINTS`
> If defined, the target supports hardware watchpoints.

`TARGET_CAN_USE_HARDWARE_WATCHPOINT` (*type*, *count*, *other*)
> Return the number of hardware watchpoints of type *type* that are possible to be set. The value is positive if *count* watchpoints of this type can be set, zero if setting watchpoints of this type is not supported, and negative if *count* is more than the maximum number of watchpoints of type *type* that can be set. *other* is non-zero if other types of watchpoints are currently enabled (there are architectures which cannot set watchpoints of different types at the same time).

`TARGET_REGION_OK_FOR_HW_WATCHPOINT` (*addr*, *len*)
> Return non-zero if hardware watchpoints can be used to watch a region whose address is *addr* and whose length in bytes is *len*.

`TARGET_REGION_SIZE_OK_FOR_HW_WATCHPOINT` (*size*)
> Return non-zero if hardware watchpoints can be used to watch a region whose size is *size*. GDB only uses this macro as a fall-back, in case `TARGET_REGION_OK_FOR_HW_WATCHPOINT` is not defined.

`TARGET_DISABLE_HW_WATCHPOINTS` (*pid*)
> Disables watchpoints in the process identified by *pid*. This is used, e.g., on HP-UX which provides operations to disable and enable the page-level memory protection that implements hardware watchpoints on that platform.

`TARGET_ENABLE_HW_WATCHPOINTS` (*pid*)
> Enables watchpoints in the process identified by *pid*. This is used, e.g., on HP-UX which provides operations to disable and enable the page-level memory protection that implements hardware watchpoints on that platform.

`target_insert_watchpoint` (*addr*, *len*, *type*)
`target_remove_watchpoint` (*addr*, *len*, *type*)

> Insert or remove a hardware watchpoint starting at *addr*, for *len* bytes. *type* is the watchpoint type, one of the possible values of the enumerated data type `target_hw_bp_type`, defined by 'breakpoint.h' as follows:
>
> ```
> enum target_hw_bp_type
>   {
>     hw_write   = 0, /* Common (write) HW watchpoint */
>     hw_read    = 1, /* Read    HW watchpoint */
>     hw_access  = 2, /* Access (read or write) HW watchpoint */
>     hw_execute = 3  /* Execute HW breakpoint */
>   };
> ```
>
> These two macros should return 0 for success, non-zero for failure.

`target_remove_hw_breakpoint` (*addr*, *shadow*)
`target_insert_hw_breakpoint` (*addr*, *shadow*)

> Insert or remove a hardware-assisted breakpoint at address *addr*. Returns zero for success, non-zero for failure. *shadow* is the real contents of the byte where the breakpoint has been inserted; it is generally not valid when hardware breakpoints are used, but since no other code touches these values, the implementations of the above two macros can use them for their internal purposes.

`target_stopped_data_address` ()

> If the inferior has some watchpoint that triggered, return the address associated with that watchpoint. Otherwise, return zero.

`DECR_PC_AFTER_HW_BREAK`

> If defined, GDB decrements the program counter by the value of `DECR_PC_AFTER_HW_BREAK` after a hardware break-point. This overrides the value of `DECR_PC_AFTER_BREAK` when a breakpoint that breaks is a hardware-assisted breakpoint.

`HAVE_STEPPABLE_WATCHPOINT`

> If defined to a non-zero value, it is not necessary to disable a watchpoint to step over it.

`HAVE_NONSTEPPABLE_WATCHPOINT`

> If defined to a non-zero value, GDB should disable a watchpoint to step the inferior over it.

`HAVE_CONTINUABLE_WATCHPOINT`

> If defined to a non-zero value, it is possible to continue the inferior after a watchpoint has been hit.

`CANNOT_STEP_HW_WATCHPOINTS`

> If this is defined to a non-zero value, GDB will remove all watchpoints before stepping the inferior.

`STOPPED_BY_WATCHPOINT` (*wait_status*)

> Return non-zero if stopped by a watchpoint. *wait_status* is of the type `struct target_waitstatus`, defined by 'target.h'.

3.8.1 x86 Watchpoints

The 32-bit Intel x86 (a.k.a. ia32) processors feature special debug registers designed to facilitate debugging. GDB provides a generic library of functions that x86-based ports can use to implement support for watchpoints and hardware-assisted breakpoints. This subsection documents the x86 watchpoint facilities in GDB.

To use the generic x86 watchpoint support, a port should do the following:

- Define the macro `I386_USE_GENERIC_WATCHPOINTS` somewhere in the target-dependent headers.
- Include the 'config/i386/nm-i386.h' header file *after* defining `I386_USE_GENERIC_WATCHPOINTS`.
- Add 'i386-nat.o' to the value of the Make variable `NATDEPFILES` (see Chapter 11 [Native Debugging], page 59) or `TDEPFILES` (see Chapter 9 [Target Architecture Definition], page 33).
- Provide implementations for the `I386_DR_LOW_*` macros described below. Typically, each macro should call a target-specific function which does the real work.

The x86 watchpoint support works by maintaining mirror images of the debug registers. Values are copied between the mirror images and the real debug registers via a set of macros which each target needs to provide:

`I386_DR_LOW_SET_CONTROL (`*val*`)`
> Set the Debug Control (DR7) register to the value *val*.

`I386_DR_LOW_SET_ADDR (`*idx,* *addr*`)`
> Put the address *addr* into the debug register number *idx*.

`I386_DR_LOW_RESET_ADDR (`*idx*`)`
> Reset (i.e. zero out) the address stored in the debug register number *idx*.

`I386_DR_LOW_GET_STATUS`
> Return the value of the Debug Status (DR6) register. This value is used immediately after it is returned by `I386_DR_LOW_GET_STATUS`, so as to support per-thread status register values.

For each one of the 4 debug registers (whose indices are from 0 to 3) that store addresses, a reference count is maintained by GDB, to allow sharing of debug registers by several watchpoints. This allows users to define several watchpoints that watch the same expression, but with different conditions and/or commands, without wasting debug registers which are in short supply. GDB maintains the reference counts internally, targets don't have to do anything to use this feature.

The x86 debug registers can each watch a region that is 1, 2, or 4 bytes long. The ia32 architecture requires that each watched region be appropriately aligned: 2-byte region on 2-byte boundary, 4-byte region on 4-byte boundary. However, the x86 watchpoint support in GDB can watch unaligned regions and regions larger than 4 bytes (up to 16 bytes) by allocating several debug registers to watch a single region. This allocation of several registers per a watched region is also done automatically without target code intervention.

The generic x86 watchpoint support provides the following API for the GDB's application code:

`i386_region_ok_for_watchpoint` (*addr, len*)

> The macro `TARGET_REGION_OK_FOR_HW_WATCHPOINT` is set to call this function. It counts the number of debug registers required to watch a given region, and returns a non-zero value if that number is less than 4, the number of debug registers available to x86 processors.

`i386_stopped_data_address` (`void`)

> The macros `STOPPED_BY_WATCHPOINT` and `target_stopped_data_address` are set to call this function. The argument passed to `STOPPED_BY_WATCHPOINT` is ignored. This function examines the breakpoint condition bits in the DR6 Debug Status register, as returned by the `I386_DR_LOW_GET_STATUS` macro, and returns the address associated with the first bit that is set in DR6.

`i386_insert_watchpoint` (*addr, len, type*)
`i386_remove_watchpoint` (*addr, len, type*)

> Insert or remove a watchpoint. The macros `target_insert_watchpoint` and `target_remove_watchpoint` are set to call these functions. `i386_insert_watchpoint` first looks for a debug register which is already set to watch the same region for the same access types; if found, it just increments the reference count of that debug register, thus implementing debug register sharing between watchpoints. If no such register is found, the function looks for a vacant debug register, sets its mirrored value to *addr*, sets the mirrored value of DR7 Debug Control register as appropriate for the *len* and *type* parameters, and then passes the new values of the debug register and DR7 to the inferior by calling `I386_DR_LOW_SET_ADDR` and `I386_DR_LOW_SET_CONTROL`. If more than one debug register is required to cover the given region, the above process is repeated for each debug register.
>
> `i386_remove_watchpoint` does the opposite: it resets the address in the mirrored value of the debug register and its read/write and length bits in the mirrored value of DR7, then passes these new values to the inferior via `I386_DR_LOW_RESET_ADDR` and `I386_DR_LOW_SET_CONTROL`. If a register is shared by several watchpoints, each time a `i386_remove_watchpoint` is called, it decrements the reference count, and only calls `I386_DR_LOW_RESET_ADDR` and `I386_DR_LOW_SET_CONTROL` when the count goes to zero.

`i386_insert_hw_breakpoint` (*addr, shadow*
`i386_remove_hw_breakpoint` (*addr, shadow*)

> These functions insert and remove hardware-assisted breakpoints. The macros `target_insert_hw_breakpoint` and `target_remove_hw_breakpoint` are set to call these functions. These functions work like `i386_insert_watchpoint` and `i386_remove_watchpoint`, respectively, except that they set up the debug registers to watch instruction execution, and each hardware-assisted breakpoint always requires exactly one debug register.

`i386_stopped_by_hwbp` (`void`)

> This function returns non-zero if the inferior has some watchpoint or hardware breakpoint that triggered. It works like `i386_stopped_data_address`, except that it doesn't return the address whose watchpoint triggered.

`i386_cleanup_dregs (void)`
> This function clears all the reference counts, addresses, and control bits in the mirror images of the debug registers. It doesn't affect the actual debug registers in the inferior process.

Notes:

1. x86 processors support setting watchpoints on I/O reads or writes. However, since no target supports this (as of March 2001), and since `enum target_hw_bp_type` doesn't even have an enumeration for I/O watchpoints, this feature is not yet available to GDB running on x86.

2. x86 processors can enable watchpoints locally, for the current task only, or globally, for all the tasks. For each debug register, there's a bit in the DR7 Debug Control register that determines whether the associated address is watched locally or globally. The current implementation of x86 watchpoint support in GDB always sets watchpoints to be locally enabled, since global watchpoints might interfere with the underlying OS and are probably unavailable in many platforms.

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.

To add paired 'set' and 'show' commands, use `add_setshow_cmd` or `add_setshow_cmd_full`. The former is a slightly simpler interface which is useful when you don't need to further modify the new command structures, while the latter returns the new command structures for manipulation.

Before removing commands from the command set it is a good idea to deprecate them for some time. Use `deprecate_cmd` on commands or aliases to set the deprecated flag. `deprecate_cmd` takes a `struct cmd_list_element` as it's first argument. You can use the return value from `add_com` or `add_cmd` to deprecate the command immediately after it is created.

The first time a command is used the user will be warned and offered a replacement (if one exists). Note that the replacement string passed to `deprecate_cmd` should be the full name of the command, i.e. the entire string the user should type at the command line.

4.2 UI-Independent Output—the `ui_out` Functions

The `ui_out` functions present an abstraction level for the GDB output code. They hide the specifics of different user interfaces supported by GDB, and thus free the programmer from the need to write several versions of the same code, one each for every UI, to produce output.

4.2.1 Overview and Terminology

In general, execution of each GDB command produces some sort of output, and can even generate an input request.

Output can be generated for the following purposes:

- to display a *result* of an operation;
- to convey *info* or produce side-effects of a requested operation;
- to provide a *notification* of an asynchronous event (including progress indication of a prolonged asynchronous operation);
- to display *error messages* (including warnings);
- to show *debug data*;
- to *query* or prompt a user for input (a special case).

This section mainly concentrates on how to build result output, although some of it also applies to other kinds of output.

Generation of output that displays the results of an operation involves one or more of the following:

- output of the actual data
- formatting the output as appropriate for console output, to make it easily readable by humans
- machine oriented formatting–a more terse formatting to allow for easy parsing by programs which read GDB's output
- annotation, whose purpose is to help legacy GUIs to identify interesting parts in the output

The `ui_out` routines take care of the first three aspects. Annotations are provided by separate annotation routines. Note that use of annotations for an interface between a GUI and GDB is deprecated.

Output can be in the form of a single item, which we call a *field*; a *list* consisting of identical fields; a *tuple* consisting of non-identical fields; or a *table*, which is a tuple consisting of a header and a body. In a BNF-like form:

```
<table> ↦
        <header> <body>

<header> ↦
        { <column> }

<column> ↦
        <width> <alignment> <title>

<body> ↦  {<row>}
```

4.2.2 General Conventions

Most `ui_out` routines are of type `void`, the exceptions are `ui_out_stream_new` (which returns a pointer to the newly created object) and the `make_cleanup` routines.

The first parameter is always the `ui_out` vector object, a pointer to a `struct ui_out`.

The *format* parameter is like in `printf` family of functions. When it is present, there must also be a variable list of arguments sufficient used to satisfy the `%` specifiers in the supplied format.

When a character string argument is not used in a `ui_out` function call, a `NULL` pointer has to be supplied instead.

4.2.3 Table, Tuple and List Functions

This section introduces `ui_out` routines for building lists, tuples and tables. The routines to output the actual data items (fields) are presented in the next section.

To recap: A *tuple* is a sequence of *fields*, each field containing information about an object; a *list* is a sequence of fields where each field describes an identical object.

Use the *table* functions when your output consists of a list of rows (tuples) and the console output should include a heading. Use this even when you are listing just one object but you still want the header.

Tables can not be nested. Tuples and lists can be nested up to a maximum of five levels.

The overall structure of the table output code is something like this:

```
ui_out_table_begin
  ui_out_table_header
  ...
  ui_out_table_body
    ui_out_tuple_begin
      ui_out_field_*
      ...
    ui_out_tuple_end
    ...
ui_out_table_end
```

Here is the description of table-, tuple- and list-related `ui_out` functions:

void **ui_out_table_begin** (struct ui_out *uiout*, int *nbrofcols*, int   *Function*
        nr_rows, const char *tblid*)

    The function `ui_out_table_begin` marks the beginning of the output of a table. It should always be called before any other `ui_out` function for a given table. *nbrofcols* is the number of columns in the table. *nr_rows* is the number of rows in the table. *tblid* is an optional string identifying the table. The string pointed to by *tblid* is copied by the implementation of `ui_out_table_begin`, so the application can free the string if it was `malloc`ed.

    The companion function `ui_out_table_end`, described below, marks the end of the table's output.

void **ui_out_table_header** (struct ui_out *_uiout_, int _width_, enum   _Function_
    ui_align _alignment_, const char *_colhdr_)

> `ui_out_table_header` provides the header information for a single table column.
> You call this function several times, one each for every column of the table, after
> `ui_out_table_begin`, but before `ui_out_table_body`.
>
> The value of _width_ gives the column width in characters. The value of _alignment_ is
> one of `left`, `center`, and `right`, and it specifies how to align the header: left-justify,
> center, or right-justify it. _colhdr_ points to a string that specifies the column header;
> the implementation copies that string, so column header strings in `malloc`ed storage
> can be freed after the call.

void **ui_out_table_body** (struct ui_out *_uiout_)   _Function_

> This function delimits the table header from the table body.

void **ui_out_table_end** (struct ui_out *_uiout_)   _Function_

> This function signals the end of a table's output. It should be called after the table
> body has been produced by the list and field output functions.
>
> There should be exactly one call to `ui_out_table_end` for each call to `ui_out_table_begin`, otherwise the `ui_out` functions will signal an internal error.

The output of the tuples that represent the table rows must follow the call to `ui_out_table_body` and precede the call to `ui_out_table_end`. You build a tuple by calling `ui_out_tuple_begin` and `ui_out_tuple_end`, with suitable calls to functions which actually output fields between them.

void **ui_out_tuple_begin** (struct ui_out *_uiout_, const char *_id_)   _Function_

> This function marks the beginning of a tuple output. _id_ points to an optional string
> that identifies the tuple; it is copied by the implementation, and so strings in `malloc`ed
> storage can be freed after the call.

void **ui_out_tuple_end** (struct ui_out *_uiout_)   _Function_

> This function signals an end of a tuple output. There should be exactly one call to
> `ui_out_tuple_end` for each call to `ui_out_tuple_begin`, otherwise an internal GDB
> error will be signaled.

struct **cleanup** *make_cleanup_ui_out_tuple_begin_end (struct   _Function_
    ui_out *_uiout_, const char *_id_)

> This function first opens the tuple and then establishes a cleanup (see Chapter 13
> [Coding], page 65) to close the tuple. It provides a convenient and correct implementation of the non-portable[1] code sequence:
>
> ```
> struct cleanup *old_cleanup;
> ui_out_tuple_begin (uiout, "...");
> old_cleanup = make_cleanup ((void(*)(void *)) ui_out_tuple_end,
>                             uiout);
> ```

---

[1] The function cast is not portable ISO C.

`void` **ui_out_list_begin** (`struct ui_out *`*uiout*`, const char *`*id*)   *Function*
> This function marks the beginning of a list output. *id* points to an optional string
> that identifies the list; it is copied by the implementation, and so strings in `malloc`ed
> storage can be freed after the call.

`void` **ui_out_list_end** (`struct ui_out *`*uiout*)   *Function*
> This function signals an end of a list output. There should be exactly one call to
> `ui_out_list_end` for each call to `ui_out_list_begin`, otherwise an internal GDB
> error will be signaled.

`struct` **cleanup** `*make_cleanup_ui_out_list_begin_end` (`struct`   *Function*
> `ui_out *`*uiout*`, const char *`*id*)
> Similar to `make_cleanup_ui_out_tuple_begin_end`, this function opens a list and
> then establishes cleanup (see Chapter 13 [Coding], page 65) that will close the list.list.

4.2.4 Item Output Functions

The functions described below produce output for the actual data items, or fields, which
contain information about the object.

Choose the appropriate function accordingly to your particular needs.

`void` **ui_out_field_fmt** (`struct ui_out *`*uiout*`, char *`*fldname*`, char`   *Function*
> `*`*format*`, ...`)
> This is the most general output function. It produces the representation of the data
> in the variable-length argument list according to formatting specifications in *format*,
> a `printf`-like format string. The optional argument *fldname* supplies the name of the
> field. The data items themselves are supplied as additional arguments after *format*.
>
> This generic function should be used only when it is not possible to use one of the
> specialized versions (see below).

`void` **ui_out_field_int** (`struct ui_out *`*uiout*`, const char *`*fldname*`,`   *Function*
> `int` *value*)
> This function outputs a value of an `int` variable. It uses the `"%d"` output conversion
> specification. *fldname* specifies the name of the field.

`void` **ui_out_field_core_addr** (`struct ui_out *`*uiout*`, const char`   *Function*
> `*`*fldname*`, CORE_ADDR` *address*)
> This function outputs an address.

`void` **ui_out_field_string** (`struct ui_out *`*uiout*`, const char`   *Function*
> `*`*fldname*`, const char *`*string*)
> This function outputs a string using the `"%s"` conversion specification.

Sometimes, there's a need to compose your output piece by piece using functions that
operate on a stream, such as `value_print` or `fprintf_symbol_filtered`. These functions
accept an argument of the type `struct ui_file *`, a pointer to a `ui_file` object used to

store the data stream used for the output. When you use one of these functions, you need a way to pass their results stored in a `ui_file` object to the `ui_out` functions. To this end, you first create a `ui_stream` object by calling `ui_out_stream_new`, pass the `stream` member of that `ui_stream` object to `value_print` and similar functions, and finally call `ui_out_field_stream` to output the field you constructed. When the `ui_stream` object is no longer needed, you should destroy it and free its memory by calling `ui_out_stream_delete`.

struct **ui_stream** *ui_out_stream_new (struct ui_out *uiout)   *Function*
> This function creates a new `ui_stream` object which uses the same output methods as the `ui_out` object whose pointer is passed in *uiout*. It returns a pointer to the newly created `ui_stream` object.

void **ui_out_stream_delete** (struct ui_stream *streambuf)   *Function*
> This functions destroys a `ui_stream` object specified by *streambuf*.

void **ui_out_field_stream** (struct ui_out *uiout, const char   *Function*
        *fieldname, struct ui_stream *streambuf)
> This function consumes all the data accumulated in `streambuf->stream` and outputs it like `ui_out_field_string` does. After a call to `ui_out_field_stream`, the accumulated data no longer exists, but the stream is still valid and may be used for producing more fields.

> **Important:** If there is any chance that your code could bail out before completing output generation and reaching the point where `ui_out_stream_delete` is called, it is necessary to set up a cleanup, to avoid leaking memory and other resources. Here's a skeleton code to do that:

```
struct ui_stream *mybuf = ui_out_stream_new (uiout);
struct cleanup *old = make_cleanup (ui_out_stream_delete, mybuf);
...
do_cleanups (old);
```

> If the function already has the old cleanup chain set (for other kinds of cleanups), you just have to add your cleanup to it:

```
mybuf = ui_out_stream_new (uiout);
make_cleanup (ui_out_stream_delete, mybuf);
```

> Note that with cleanups in place, you should not call `ui_out_stream_delete` directly, or you would attempt to free the same buffer twice.

4.2.5 Utility Output Functions

void **ui_out_field_skip** (struct ui_out *uiout, const char *fldname)   *Function*

> This function skips a field in a table. Use it if you have to leave an empty field without disrupting the table alignment. The argument *fldname* specifies a name for the (missing) filed.

`void` **ui_out_text** (`struct ui_out *`*uiout*`, const char *`*string*)   Function
> This function outputs the text in *string* in a way that makes it easy to be read by humans. For example, the console implementation of this method filters the text through a built-in pager, to prevent it from scrolling off the visible portion of the screen.
>
> Use this function for printing relatively long chunks of text around the actual field data: the text it produces is not aligned according to the table's format. Use `ui_out_field_string` to output a string field, and use `ui_out_message`, described below, to output short messages.

`void` **ui_out_spaces** (`struct ui_out *`*uiout*`, int` *nspaces*)   Function
> This function outputs *nspaces* spaces. It is handy to align the text produced by `ui_out_text` with the rest of the table or list.

`void` **ui_out_message** (`struct ui_out *`*uiout*`, int` *verbosity*`, const`   Function
>   `char *`*format*`, ...`)
> This function produces a formatted message, provided that the current verbosity level is at least as large as given by *verbosity*. The current verbosity level is specified by the user with the '`set verbositylevel`' command.[2]

`void` **ui_out_wrap_hint** (`struct ui_out *`*uiout*`, char *`*indent*)   Function
> This function gives the console output filter (a paging filter) a hint of where to break lines which are too long. Ignored for all other output consumers. *indent*, if non-`NULL`, is the string to be printed to indent the wrapped text on the next line; it must remain accessible until the next call to `ui_out_wrap_hint`, or until an explicit newline is produced by one of the other functions. If *indent* is `NULL`, the wrapped text will not be indented.

`void` **ui_out_flush** (`struct ui_out *`*uiout*)   Function
> This function flushes whatever output has been accumulated so far, if the UI buffers output.

4.2.6 Examples of Use of `ui_out` functions

This section gives some practical examples of using the `ui_out` functions to generalize the old console-oriented code in GDB. The examples all come from functions defined on the '`breakpoints.c`' file.

This example, from the `breakpoint_1` function, shows how to produce a table.

The original code was:

```
if (!found_a_breakpoint++)
  {
    annotate_breakpoints_headers ();
```

---

[2] As of this writing (April 2001), setting verbosity level is not yet implemented, and is always returned as zero. So calling `ui_out_message` with a *verbosity* argument more than zero will cause the message to never be printed.

```
      annotate_field (0);
      printf_filtered ("Num ");
      annotate_field (1);
      printf_filtered ("Type           ");
      annotate_field (2);
      printf_filtered ("Disp ");
      annotate_field (3);
      printf_filtered ("Enb ");
      if (addressprint)
        {
          annotate_field (4);
          printf_filtered ("Address    ");
        }
      annotate_field (5);
      printf_filtered ("What\n");

      annotate_breakpoints_table ();
    }
```

Here's the new version:

```
    nr_printable_breakpoints = ...;

    if (addressprint)
      ui_out_table_begin (ui, 6, nr_printable_breakpoints, "BreakpointTable");
    else
      ui_out_table_begin (ui, 5, nr_printable_breakpoints, "BreakpointTable");

    if (nr_printable_breakpoints > 0)
      annotate_breakpoints_headers ();
    if (nr_printable_breakpoints > 0)
      annotate_field (0);
    ui_out_table_header (uiout, 3, ui_left, "number", "Num"); /* 1 */
    if (nr_printable_breakpoints > 0)
      annotate_field (1);
    ui_out_table_header (uiout, 14, ui_left, "type", "Type"); /* 2 */
    if (nr_printable_breakpoints > 0)
      annotate_field (2);
    ui_out_table_header (uiout, 4, ui_left, "disp", "Disp"); /* 3 */
    if (nr_printable_breakpoints > 0)
      annotate_field (3);
    ui_out_table_header (uiout, 3, ui_left, "enabled", "Enb"); /* 4 */
    if (addressprint)
      {
       if (nr_printable_breakpoints > 0)
         annotate_field (4);
       if (TARGET_ADDR_BIT <= 32)
         ui_out_table_header (uiout, 10, ui_left, "addr", "Address");/* 5 */
       else
         ui_out_table_header (uiout, 18, ui_left, "addr", "Address");/* 5 */
```

```
          }
        if (nr_printable_breakpoints > 0)
          annotate_field (5);
        ui_out_table_header (uiout, 40, ui_noalign, "what", "What"); /* 6 */
        ui_out_table_body (uiout);
        if (nr_printable_breakpoints > 0)
          annotate_breakpoints_table ();
```

This example, from the `print_one_breakpoint` function, shows how to produce the
actual data for the table whose structure was defined in the above example. The original
code was:

```
        annotate_record ();
        annotate_field (0);
        printf_filtered ("%-3d ", b->number);
        annotate_field (1);
        if ((int)b->type > (sizeof(bptypes)/sizeof(bptypes[0]))
            || ((int) b->type != bptypes[(int) b->type].type))
          internal_error ("bptypes table does not describe type #%d.",
                          (int)b->type);
        printf_filtered ("%-14s ", bptypes[(int)b->type].description);
        annotate_field (2);
        printf_filtered ("%-4s ", bpdisps[(int)b->disposition]);
        annotate_field (3);
        printf_filtered ("%-3c ", bpenables[(int)b->enable]);
        ...
```

This is the new version:

```
        annotate_record ();
        ui_out_tuple_begin (uiout, "bkpt");
        annotate_field (0);
        ui_out_field_int (uiout, "number", b->number);
        annotate_field (1);
        if (((int) b->type > (sizeof (bptypes) / sizeof (bptypes[0])))
            || ((int) b->type != bptypes[(int) b->type].type))
          internal_error ("bptypes table does not describe type #%d.",
                          (int) b->type);
        ui_out_field_string (uiout, "type", bptypes[(int)b->type].description);
        annotate_field (2);
        ui_out_field_string (uiout, "disp", bpdisps[(int)b->disposition]);
        annotate_field (3);
        ui_out_field_fmt (uiout, "enabled", "%c", bpenables[(int)b->enable]);
        ...
```

This example, also from `print_one_breakpoint`, shows how to produce a complicated
output field using the `print_expression` functions which requires a stream to be passed.
It also shows how to automate stream destruction with cleanups. The original code was:

```
        annotate_field (5);
        print_expression (b->exp, gdb_stdout);
```

The new version is:

```
        struct ui_stream *stb = ui_out_stream_new (uiout);
```

```
      struct cleanup *old_chain = make_cleanup_ui_out_stream_delete (stb);
      ...
      annotate_field (5);
      print_expression (b->exp, stb->stream);
      ui_out_field_stream (uiout, "what", local_stream);
```

This example, also from `print_one_breakpoint`, shows how to use `ui_out_text` and `ui_out_field_string`. The original code was:

```
      annotate_field (5);
      if (b->dll_pathname == NULL)
        printf_filtered ("<any library> ");
      else
        printf_filtered ("library \"%s\" ", b->dll_pathname);
```

It became:

```
      annotate_field (5);
      if (b->dll_pathname == NULL)
        {
          ui_out_field_string (uiout, "what", "<any library>");
          ui_out_spaces (uiout, 1);
        }
      else
        {
          ui_out_text (uiout, "library \"");
          ui_out_field_string (uiout, "what", b->dll_pathname);
          ui_out_text (uiout, "\" ");
        }
```

The following example from `print_one_breakpoint` shows how to use `ui_out_field_int` and `ui_out_spaces`. The original code was:

```
      annotate_field (5);
      if (b->forked_inferior_pid != 0)
        printf_filtered ("process %d ", b->forked_inferior_pid);
```

It became:

```
      annotate_field (5);
      if (b->forked_inferior_pid != 0)
        {
          ui_out_text (uiout, "process ");
          ui_out_field_int (uiout, "what", b->forked_inferior_pid);
          ui_out_spaces (uiout, 1);
        }
```

Here's an example of using `ui_out_field_string`. The original code was:

```
      annotate_field (5);
      if (b->exec_pathname != NULL)
        printf_filtered ("program \"%s\" ", b->exec_pathname);
```

It became:

```
      annotate_field (5);
      if (b->exec_pathname != NULL)
        {
```

```
        ui_out_text (uiout, "program \"");
        ui_out_field_string (uiout, "what", b->exec_pathname);
        ui_out_text (uiout, "\" ");
      }
```

Finally, here's an example of printing an address. The original code:

```
    annotate_field (4);
    printf_filtered ("%s ",
        local_hex_string_custom ((unsigned long) b->address, "08l"));
```

It became:

```
    annotate_field (4);
    ui_out_field_core_addr (uiout, "Address", b->address);
```

4.3 Console Printing

4.4 TUI

5 libgdb

5.1 libgdb 1.0

`libgdb` 1.0 was an abortive project of years ago. The theory was to provide an API to GDB's functionality.

5.2 libgdb 2.0

`libgdb` 2.0 is an ongoing effort to update GDB so that is better able to support graphical and other environments.

Since `libgdb` development is on-going, its architecture is still evolving. The following components have so far been identified:

- Observer - 'gdb-events.h'.
- Builder - 'ui-out.h'
- Event Loop - 'event-loop.h'
- Library - 'gdb.h'

The model that ties these components together is described below.

5.3 The `libgdb` Model

A client of `libgdb` interacts with the library in two ways.

- As an observer (using '`gdb-events`') receiving notifications from `libgdb` of any internal state changes (break point changes, run state, etc).

- As a client querying `libgdb` (using the '`ui-out`' builder) to obtain various status values from GDB.

Since `libgdb` could have multiple clients (e.g. a GUI supporting the existing GDB CLI), those clients must co-operate when controlling `libgdb`. In particular, a client must ensure that `libgdb` is idle (i.e. no other client is using `libgdb`) before responding to a '`gdb-event`' by making a query.

5.4 CLI support

At present GDB's CLI is very much entangled in with the core of `libgdb`. Consequently, a client wishing to include the CLI in their interface needs to carefully co-ordinate its own and the CLI's requirements.

It is suggested that the client set `libgdb` up to be bi-modal (alternate between CLI and client query modes). The notes below sketch out the theory:

- The client registers itself as an observer of `libgdb`.

- The client create and install `cli-out` builder using its own versions of the `ui-file` `gdb_stderr`, `gdb_stdtarg` and `gdb_stdout` streams.

- The client creates a separate custom `ui-out` builder that is only used while making direct queries to `libgdb`.

When the client receives input intended for the CLI, it simply passes it along. Since the `cli-out` builder is installed by default, all the CLI output in response to that command is routed (pronounced rooted) through to the client controlled `gdb_stdout` et. al. streams. At the same time, the client is kept abreast of internal changes by virtue of being a `libgdb` observer.

The only restriction on the client is that it must wait until `libgdb` becomes idle before initiating any queries (using the client's custom builder).

5.5 `libgdb` components

Observer - '`gdb-events.h`'

'`gdb-events`' provides the client with a very raw mechanism that can be used to implement an observer. At present it only allows for one observer and that observer must, internally, handle the need to delay the processing of any event notifications until after `libgdb` has finished the current command.

Builder - '`ui-out.h`'

'`ui-out`' provides the infrastructure necessary for a client to create a builder. That builder is then passed down to `libgdb` when doing any queries.

Event Loop - 'event-loop.h'

'event-loop', currently non-re-entrant, provides a simple event loop. A client would need to either plug its self into this loop or, implement a new event-loop that GDB would use.

The event-loop will eventually be made re-entrant. This is so that [No value for "GDB"] can better handle the problem of some commands blocking instead of returning.

Library - 'gdb.h'

'libgdb' is the most obvious component of this system. It provides the query interface. Each function is parameterized by a ui-out builder. The result of the query is constructed using that builder before the query function returns.

6 Symbol Handling

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

6.1 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 (see Chapter 12 [Support Libraries], page 63). BFD identifies the type of the file by examining its header. find_sym_fns 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 needs 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.

6.2 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.

6.3 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.

6.4 Object File Formats

6.4.1 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'.

6.4.2 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'.

6.4.3 ECOFF

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

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

6.4.4 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 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).

6.4.5 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.

6.4.6 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'.

6.4.7 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'.

6.4.8 Other File Formats

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

6.5 Debugging File Formats

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

6.5.1 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.

6.5.2 COFF

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

6.5.3 Mips debug (Third Eye)

ECOFF includes a definition of a special debug format.

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

6.5.4 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'.

6.5.5 DWARF 2

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

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

6.5.6 SOM

Like COFF, the SOM definition includes debugging information.

6.6 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/lib*xyz*.h', which is included by GDB.

7 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, like with DWARF.

7.1 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 the `evaluate_subexp` function defined in the file '`eval.c`'. Add cases for new opcodes in two functions from '`parse.c`': `prefixify_subexp` and

`length_of_subexp`. These compute the number of `exp_element`s 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 `print_subexp` (in 'expprint.c') 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_exp_1` (defined in 'parse.c').

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!

8 Host Definition

With the advent of Autoconf, it's rarely necessary to have host definition machinery anymore. The following information is provided, mainly, as an historical reference.

8.1 Adding a New Host

GDB's host configuration support normally happens via Autoconf. New host-specific definitions should not be needed. Older hosts GDB still use the host-specific definitions and files listed below, but these mostly exist for historical reasons, and will eventually disappear.

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

> This file once contained both host and native configuration information (see Chapter 11 [Native Debugging], page 59) for the machine *xyz*. The host configuration information is now handed by Autoconf.
>
> Host configuration information included a definition of XM_FILE=xm-*xyz*.h and possibly definitions for CC, SYSV_DEFINE, XM_CFLAGS, XM_ADD_FILES, XM_CLIBS, XM_CDEPS, etc.; see 'Makefile.in'.
>
> New host only configurations do not need this file.

'gdb/config/*arch*/xm-*xyz*.h'

> This file once contained definitions and includes required when hosting gdb on machine *xyz*. Those definitions and includes are now handled by Autoconf.
>
> New host and native configurations do not need this file.
>
> *Maintainer's note: Some hosts continue to use the 'xm-xyz.h' file to define the macros HOST_FLOAT_FORMAT, HOST_DOUBLE_FORMAT and HOST_LONG_DOUBLE_FORMAT. That code also needs to be replaced with either an Autoconf or run-time test.*

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 DJGPP (a.k.a. GO32) execution environment.

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

8.2 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`').

`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_TERMIO
>   Define this if the host system has `termio.h`.

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 conversion specifier `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 conversion specifier `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 conversion specifier `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 equiv-
>   alent.

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.

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.

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 `/**/`.

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.

The target architecture object is implemented as the C structure `struct gdbarch *`.
The structure, and its methods, are generated using the Bourne shell script 'gdbarch.sh'.

9.1 Operating System ABI Variant Handling

GDB provides a mechanism for handling variations in OS ABIs. An OS ABI variant
may have influence over any number of variables in the target architecture definition. There
are two major components in the OS ABI mechanism: sniffers and handlers.

A *sniffer* examines a file matching a BFD architecture/flavour pair (the architecture
may be wildcarded) in an attempt to determine the OS ABI of that file. Sniffers with a
wildcarded architecture are considered to be *generic*, while sniffers for a specific architecture
are considered to be *specific*. A match from a specific sniffer overrides a match from a generic
sniffer. Multiple sniffers for an architecture/flavour may exist, in order to differentiate
between two different operating systems which use the same basic file format. The OS ABI
framework provides a generic sniffer for ELF-format files which examines the `EI_OSABI` field
of the ELF header, as well as note sections known to be used by several operating systems.

A *handler* is used to fine-tune the `gdbarch` structure for the selected OS ABI. There
may be only one handler for a given OS ABI for each BFD architecture.

The following OS ABI variants are defined in 'osabi.h':

GDB_OSABI_UNKNOWN
          The ABI of the inferior is unknown. The default `gdbarch` settings for the
          architecture will be used.

GDB_OSABI_SVR4
          UNIX System V Release 4

GDB_OSABI_HURD
          GNU using the Hurd kernel

GDB_OSABI_SOLARIS
          Sun Solaris

GDB_OSABI_OSF1
> OSF/1, including Digital UNIX and Compaq Tru64 UNIX

GDB_OSABI_LINUX
> GNU using the Linux kernel

GDB_OSABI_FREEBSD_AOUT
> FreeBSD using the a.out executable format

GDB_OSABI_FREEBSD_ELF
> FreeBSD using the ELF executable format

GDB_OSABI_NETBSD_AOUT
> NetBSD using the a.out executable format

GDB_OSABI_NETBSD_ELF
> NetBSD using the ELF executable format

GDB_OSABI_WINCE
> Windows CE

GDB_OSABI_GO32
> DJGPP

GDB_OSABI_NETWARE
> Novell NetWare

GDB_OSABI_ARM_EABI_V1
> ARM Embedded ABI version 1

GDB_OSABI_ARM_EABI_V2
> ARM Embedded ABI version 2

GDB_OSABI_ARM_APCS
> Generic ARM Procedure Call Standard

Here are the functions that make up the OS ABI framework:

const char *gdbarch_osabi_name (enum gdb_osabi *osabi*)   *Function*
> Return the name of the OS ABI corresponding to *osabi*.

void **gdbarch_register_osabi** (enum bfd_architecture *arch*, enum   *Function*
> gdb_osabi *osabi*, void (**init_osabi*)(struct gdbarch_info *info*, struct
> gdbarch **gdbarch*))
> Register the OS ABI handler specified by *init_osabi* for the architecture/OS ABI pair
> specified by *arch* and *osabi*.

void **gdbarch_register_osabi_sniffer** (enum bfd_architecture   *Function*
> *arch*, enum bfd_flavour *flavour*, enum gdb_osabi (**sniffer*)(bfd **abfd*))
> Register the OS ABI file sniffer specified by *sniffer* for the BFD architecture/flavour
> pair specified by *arch* and *flavour*. If *arch* is bfd_arch_unknown, the sniffer is consid-
> ered to be generic, and is allowed to examine *flavour*-flavoured files for any architec-
> ture.

enum **gdb_osabi** `gdbarch_lookup_osabi` (`bfd *`*abfd*)   *Function*
    Examine the file described by *abfd* to determine its OS ABI. The value `GDB_OSABI_`
    `UNKNOWN` is returned if the OS ABI cannot be determined.

void **gdbarch_init_osabi** (`struct gdbarch info` *info*, `struct`   *Function*
        `gdbarch *`*gdbarch*, `enum gdb_osabi` *osabi*)
    Invoke the OS ABI handler corresponding to *osabi* to fine-tune the `gdbarch` structure
    specified by *gdbarch*. If a handler corresponding to *osabi* has not been registered
    for *gdbarch*'s architecture, a warning will be issued and the debugging session will
    continue with the defaults already established for *gdbarch*.

9.2 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.

9.3 Pointers Are Not Always Addresses

On almost all 32-bit architectures, the representation of a pointer is indistinguishable
from the representation of some fixed-length number whose value is the byte address of
the object pointed to. On such machines, the words "pointer" and "address" can be used
interchangeably. However, architectures with smaller word sizes are often cramped for
address space, so they may choose a pointer representation that breaks this identity, and
allows a larger code address space.

For example, the Mitsubishi D10V is a 16-bit VLIW processor whose instructions are
32 bits long[3]. If the D10V used ordinary byte addresses to refer to code locations, then the
processor would only be able to address 64kb of instructions. However, since instructions
must be aligned on four-byte boundaries, the low two bits of any valid instruction's byte
address are always zero—byte addresses waste two bits. So instead of byte addresses, the
D10V uses word addresses—byte addresses shifted right two bits—to refer to code. Thus,
the D10V can use 16-bit words to address 256kb of code space.

However, this means that code pointers and data pointers have different forms on the
D10V. The 16-bit word `0xC020` refers to byte address `0xC020` when used as a data address,
but refers to byte address `0x30080` when used as a code address.

(The D10V also uses separate code and data address spaces, which also affects the
correspondence between pointers and addresses, but we're going to ignore that here; this
example is already too long.)

---

[3] Some D10V instructions are actually pairs of 16-bit sub-instructions. However, since you can't jump
    into the middle of such a pair, code addresses can only refer to full 32 bit instructions, which is what
    matters in this explanation.

To cope with architectures like this—the D10V is not the only one!—GDB tries to distinguish between *addresses*, which are byte numbers, and *pointers*, which are the target's representation of an address of a particular type of data. In the example above, `0xC020` is the pointer, which refers to one of the addresses `0xC020` or `0x30080`, depending on the type imposed upon it. GDB provides functions for turning a pointer into an address and vice versa, in the appropriate way for the current architecture.

Unfortunately, since addresses and pointers are identical on almost all processors, this distinction tends to bit-rot pretty quickly. Thus, each time you port GDB to an architecture which does distinguish between pointers and addresses, you'll probably need to clean up some architecture-independent code.

Here are functions which convert between pointers and addresses:

CORE_ADDR **extract_typed_address** (void **buf*, struct type **type*)   Function
> Treat the bytes at *buf* as a pointer or reference of type *type*, and return the address it represents, in a manner appropriate for the current architecture. This yields an address GDB can use to read target memory, disassemble, etc. Note that *buf* refers to a buffer in GDB's memory, not the inferior's.
>
> For example, if the current architecture is the Intel x86, this function extracts a little-endian integer of the appropriate length from *buf* and returns it. However, if the current architecture is the D10V, this function will return a 16-bit integer extracted from *buf*, multiplied by four if *type* is a pointer to a function.
>
> If *type* is not a pointer or reference type, then this function will signal an internal error.

CORE_ADDR **store_typed_address** (void **buf*, struct type **type*,   Function
>   CORE_ADDR *addr*)
> Store the address *addr* in *buf*, in the proper format for a pointer of type *type* in the current architecture. Note that *buf* refers to a buffer in GDB's memory, not the inferior's.
>
> For example, if the current architecture is the Intel x86, this function stores *addr* unmodified as a little-endian integer of the appropriate length in *buf*. However, if the current architecture is the D10V, this function divides *addr* by four if *type* is a pointer to a function, and then stores it in *buf*.
>
> If *type* is not a pointer or reference type, then this function will signal an internal error.

CORE_ADDR **value_as_address** (struct value **val*)   Function
> Assuming that *val* is a pointer, return the address it represents, as appropriate for the current architecture.
>
> This function actually works on integral values, as well as pointers. For pointers, it performs architecture-specific conversions as described above for `extract_typed_address`.

CORE_ADDR **value_from_pointer** (`struct type *`*type*, `CORE_ADDR`   *Function*
      addr)

      Create and return a value representing a pointer of type *type* to the address *addr*, as appropriate for the current architecture. This function performs architecture-specific conversions as described above for `store_typed_address`.

GDB also provides functions that do the same tasks, but assume that pointers are simply byte addresses; they aren't sensitive to the current architecture, beyond knowing the appropriate endianness.

CORE_ADDR **extract_address** (`void *`*addr*, `int len`)   *Function*

      Extract a *len*-byte number from *addr* in the appropriate endianness for the current architecture, and return it. Note that *addr* refers to GDB's memory, not the inferior's.

      This function should only be used in architecture-specific code; it doesn't have enough information to turn bits into a true address in the appropriate way for the current architecture. If you can, use `extract_typed_address` instead.

void **store_address** (`void *`*addr*, `int `*len*, `LONGEST `*val*)   *Function*

      Store *val* at *addr* as a *len*-byte integer, in the appropriate endianness for the current architecture. Note that *addr* refers to a buffer in GDB's memory, not the inferior's.

      This function should only be used in architecture-specific code; it doesn't have enough information to turn a true address into bits in the appropriate way for the current architecture. If you can, use `store_typed_address` instead.

Here are some macros which architectures can define to indicate the relationship between pointers and addresses. These have default definitions, appropriate for architectures on which all pointers are simple unsigned byte addresses.

CORE_ADDR **POINTER_TO_ADDRESS** (`struct type *`*type*,   *Target Macro*
      `char *`*buf*)

      Assume that *buf* holds a pointer of type *type*, in the appropriate format for the current architecture. Return the byte address the pointer refers to.

      This function may safely assume that *type* is either a pointer or a C++ reference type.

void **ADDRESS_TO_POINTER** (`struct type *`*type*, `char`   *Target Macro*
      `*`*buf*, `CORE_ADDR `*addr*)

      Store in *buf* a pointer of type *type* representing the address *addr*, in the appropriate format for the current architecture.

      This function may safely assume that *type* is either a pointer or a C++ reference type.

9.4 Raw and Virtual Register Representations

Maintainer note: This section is pretty much obsolete. The functionality described here has largely been replaced by pseudo-registers and the mechanisms described in Chapter 9 [Using Different Register and Memory Data Representations], page 33. See also Bug Tracking Database and ARI Index for more up-to-date information.

Some architectures use one representation for a value when it lives in a register, but use a different representation when it lives in memory. In GDB's terminology, the *raw* representation is the one used in the target registers, and the *virtual* representation is the one used in memory, and within GDB `struct value` objects.

Maintainer note: Notice that the same mechanism is being used to both convert a register to a `struct value` *and alternative register forms.*

For almost all data types on almost all architectures, the virtual and raw representations are identical, and no special handling is needed. However, they do occasionally differ. For example:

- The x86 architecture supports an 80-bit `long double` type. However, when we store those values in memory, they occupy twelve bytes: the floating-point number occupies the first ten, and the final two bytes are unused. This keeps the values aligned on four-byte boundaries, allowing more efficient access. Thus, the x86 80-bit floating-point type is the raw representation, and the twelve-byte loosely-packed arrangement is the virtual representation.

- Some 64-bit MIPS targets present 32-bit registers to GDB as 64-bit registers, with garbage in their upper bits. GDB ignores the top 32 bits. Thus, the 64-bit form, with garbage in the upper 32 bits, is the raw representation, and the trimmed 32-bit representation is the virtual representation.

In general, the raw representation is determined by the architecture, or GDB's interface to the architecture, while the virtual representation can be chosen for GDB's convenience. GDB's register file, `registers`, holds the register contents in raw format, and the GDB remote protocol transmits register values in raw format.

Your architecture may define the following macros to request conversions between the raw and virtual format:

---

int **REGISTER_CONVERTIBLE** (int *reg*)   Target Macro
    Return non-zero if register number *reg*'s value needs different raw and virtual formats.

    You should not use `REGISTER_CONVERT_TO_VIRTUAL` for a register unless this macro returns a non-zero value for that register.

---

int **REGISTER_RAW_SIZE** (int *reg*)   Target Macro
    The size of register number *reg*'s raw value. This is the number of bytes the register will occupy in `registers`, or in a GDB remote protocol packet.

---

int **REGISTER_VIRTUAL_SIZE** (int *reg*)   Target Macro
    The size of register number *reg*'s value, in its virtual format. This is the size a `struct value`'s buffer will have, holding that register's value.

---

struct type *`REGISTER_VIRTUAL_TYPE` (int *reg*)   Target Macro
    This is the type of the virtual representation of register number *reg*. Note that there is no need for a macro giving a type for the register's raw form; once the register's value has been obtained, GDB always uses the virtual form.

`void` **REGISTER_CONVERT_TO_VIRTUAL** (`int` *reg*,   *Target Macro*
      `struct type *`*type*`, char *`*from*`, char *`*to*)

> Convert the value of register number *reg* to *type*, which should always be `REGISTER_VIRTUAL_TYPE` (*reg*). The buffer at *from* holds the register's value in raw format; the macro should convert the value to virtual format, and place it at *to*.

> Note that `REGISTER_CONVERT_TO_VIRTUAL` and `REGISTER_CONVERT_TO_RAW` take their *reg* and *type* arguments in different orders.

> You should only use `REGISTER_CONVERT_TO_VIRTUAL` with registers for which the `REGISTER_CONVERTIBLE` macro returns a non-zero value.

`void` **REGISTER_CONVERT_TO_RAW** (`struct type`   *Target Macro*
      `*`*type*`, int` *reg*`, char *`*from*`, char *`*to*)

> Convert the value of register number *reg* to *type*, which should always be `REGISTER_VIRTUAL_TYPE` (*reg*). The buffer at *from* holds the register's value in raw format; the macro should convert the value to virtual format, and place it at *to*.

> Note that REGISTER_CONVERT_TO_VIRTUAL and REGISTER_CONVERT_TO_RAW take their *reg* and *type* arguments in different orders.

9.5 Using Different Register and Memory Data Representations

Maintainer's note: The way GDB manipulates registers is undergoing significant change. Many of the macros and functions refered to in this section are likely to be subject to further revision. See A.R. Index and Bug Tracking Database for further information. cagney/2002-05-06.

Some architectures can represent a data object in a register using a form that is different to the objects more normal memory representation. For example:

- The Alpha architecture can represent 32 bit integer values in floating-point registers.
- The x86 architecture supports 80-bit floating-point registers. The `long double` data type occupies 96 bits in memory but only 80 bits when stored in a register.

In general, the register representation of a data type is determined by the architecture, or GDB's interface to the architecture, while the memory representation is determined by the Application Binary Interface.

For almost all data types on almost all architectures, the two representations are identical, and no special handling is needed. However, they do occasionally differ. Your architecture may define the following macros to request conversions between the register and memory representations of a data type:

`int` **CONVERT_REGISTER_P** (`int` *reg*)   *Target Macro*

> Return non-zero if the representation of a data value stored in this register may be different to the representation of that same data value when stored in memory.

> When non-zero, the macros `REGISTER_TO_VALUE` and `VALUE_TO_REGISTER` are used to perform any necessary conversion.

void **REGISTER_TO_VALUE** (`int` *reg*, `struct type *`*type*,   *Target Macro*
       `char *`*from*, `char *`*to*)

> Convert the value of register number *reg* to a data object of type *type*. The buffer at *from* holds the register's value in raw format; the converted value should be placed in the buffer at *to*.
>
> Note that `REGISTER_TO_VALUE` and `VALUE_TO_REGISTER` take their *reg* and *type* arguments in different orders.
>
> You should only use `REGISTER_TO_VALUE` with registers for which the `CONVERT_REGISTER_P` macro returns a non-zero value.

void **VALUE_TO_REGISTER** (`struct type *`*type*, `int` *reg*,   *Target Macro*
       `char *`*from*, `char *`*to*)

> Convert a data value of type *type* to register number *reg*' raw format.
>
> Note that `REGISTER_TO_VALUE` and `VALUE_TO_REGISTER` take their *reg* and *type* arguments in different orders.
>
> You should only use `VALUE_TO_REGISTER` with registers for which the `CONVERT_REGISTER_P` macro returns a non-zero value.

void **REGISTER_CONVERT_TO_TYPE** (`int` *regnum*,   *Target Macro*
       `struct type *`*type*, `char *`*buf*)

> See '`mips-tdep.c`'. It does not do what you want.

9.6 Frame Interpretation

9.7 Inferior Call Setup

9.8 Compiler Characteristics

9.9 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 instruction address includes any bits that are not really part of the address, then define this macro to expand into an expression that zeroes

those bits in *addr*. This is only used for addresses of instructions, and even then not in all contexts.

For example, the two low-order bits of the PC on the Hewlett-Packard PA 2.0 architecture contain the privilege level of the corresponding instruction. Since instructions must always be aligned on four-byte boundaries, the processor masks out these bits to generate the actual address of the instruction. ADDR_BITS_REMOVE should filter out these bits with an expression such as `((addr) & ~3)`.

ADDRESS_TO_POINTER (*type, buf, addr*)

Store in *buf* a pointer of type *type* representing the address *addr*, in the appropriate format for the current architecture. This macro may safely assume that *type* is either a pointer or a C++ reference type. See Chapter 9 [Pointers Are Not Always Addresses], page 33.

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 macro 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 bit 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 favor 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 favor 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 favor 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.

`MEMORY_INSERT_BREAKPOINT` (*addr*, *contents_cache*)
`MEMORY_REMOVE_BREAKPOINT` (*addr*, *contents_cache*)

> Insert or remove memory based breakpoints. Reasonable defaults (`default_memory_insert_breakpoint` and `default_memory_remove_breakpoint` respectively) have been provided so that it is not necessary to define these for most architectures. Architectures which may want to define `MEMORY_INSERT_BREAKPOINT` and `MEMORY_REMOVE_BREAKPOINT` will likely have instructions that are oddly sized or are not stored in a conventional manner.
>
> It may also be desirable (from an efficiency standpoint) to define custom breakpoint insertion and removal routines if `BREAKPOINT_FROM_PC` needs to read the target's memory for some reason.

`CALL_DUMMY_P`

> A C expression that is non-zero when the target supports 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 favor 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`

> See the file '`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?

`COERCE_FLOAT_TO_DOUBLE` (*formal, actual*)

Return non-zero if GDB should promote `float` values to `double` when calling a non-prototyped function. The argument *actual* is the type of the value we want to pass to the function. The argument *formal* is the type of this argument, as it appears in the function's definition. Note that *formal* may be zero if we have no debugging information for the function, or if we're passing more arguments than are officially declared (for example, varargs). This macro is never invoked if the function definitely has a prototype.

How you should pass arguments to a function depends on whether it was defined in K&R style or prototype style. If you define a function using the K&R syntax that takes a `float` argument, then callers must pass that argument as a `double`. If you define the function using the prototype syntax, then you must pass the argument as a `float`, with no promotion.

Unfortunately, on certain older platforms, the debug info doesn't indicate reliably how each function was defined. A function type's `TYPE_FLAG_PROTOTYPED` flag may be unset, even if the function was defined in prototype style. When calling a function whose `TYPE_FLAG_PROTOTYPED` flag is unset, GDB consults the `COERCE_FLOAT_TO_DOUBLE` macro to decide what to do.

For modern targets, it is proper to assume that, if the prototype flag is unset, that can be trusted: `float` arguments should be promoted to `double`. You should use the function `standard_coerce_float_to_double` to get this behavior.

For some older targets, if the prototype flag is unset, that doesn't tell us anything. So we guess that, if we don't have a type for the formal parameter (*i.e.*, the first argument to `COERCE_FLOAT_TO_DOUBLE` is null), then we should

promote it; otherwise, we should leave it alone. The function `default_coerce_float_to_double` provides this behavior; it is the default value, for compatibility with older configurations.

`int CONVERT_REGISTER_P(`*regnum*`)`

Return non-zero if register *regnum* can represent data values in a non-standard form. See Chapter 9 [Using Different Register and Memory Data Representations], page 33.

`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.

This method is deprecated.

`PRINT_FLOAT_INFO()`

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

`print_registers_info (`*gdbarch*`, `*frame*`, `*regnum*`, `*all*`)`

If defined, pretty print the value of the register *regnum* for the specified *frame*. If the value of *regnum* is -1, pretty print either all registers (*all* is non zero) or a select subset of registers (*all* is zero).

The default method prints one register per line, and if *all* is zero omits floating-point registers.

`PRINT_VECTOR_INFO()`

If defined, then the '`info vector`' command will call this function to print information about the processor's vector unit.

By default, the '`info vector`' command will print all vector registers (the register's type having the vector attribute).

`DWARF_REG_TO_REGNUM`
> Convert DWARF register number into GDB regnum. If not defined, no conversion will be performed.

`DWARF2_REG_TO_REGNUM`
> Convert DWARF2 register number into GDB regnum. If not defined, no conversion will be performed.

`ECOFF_REG_TO_REGNUM`
> Convert ECOFF register number into GDB regnum. If not defined, no conversion will be performed.

`END_OF_TEXT_DEFAULT`
> This is an expression that should designate the end of the text section.

`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 defined, extract from the array *regbuf* (containing the raw register state) the `CORE_ADDR` at which a function should return its structure value.
>
> If not defined, `EXTRACT_RETURN_VALUE` is used.

`EXTRACT_STRUCT_VALUE_ADDRESS_P()`
> Predicate for `EXTRACT_STRUCT_VALUE_ADDRESS`.

`FLOAT_INFO`
> Deprecated in favor of `PRINT_FLOAT_INFO`.

`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` is 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`
> See '`stack.c`'.

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

`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. Several common definitions are available:
> - `file_frame_chain_valid` is nonzero if the chain pointer is nonzero and given frame's PC is not inside the startup file (such as '`crt0.o`').
> - `func_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`).

- `generic_file_frame_chain_valid` and `generic_func_frame_chain_valid` are equivalent implementations for targets using generic dummy frames.

`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. This 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.

`FUNCTION_START_OFFSET`

> An integer, giving the offset in bytes from a function's address (as used in the values of symbols, function pointers, etc.), and the function's first genuine instruction.
>
> This is zero on almost all machines: the function's address is usually the address of its first instruction. However, on the VAX, for example, each function starts with two bytes containing a bitmask indicating which registers to save upon entry to the function. The VAX `call` instructions check this value, and save the appropriate registers automatically. Thus, since the offset from the function's address to its first instruction is two bytes, `FUNCTION_START_OFFSET` would be 2 on the VAX.

`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 support for multiple architectures within GDB.
>
> This 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 dependent 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 instead.

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 the header file '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.

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 a haste and we are repenting at leisure.

I386_USE_GENERIC_WATCHPOINTS

> An x86-based target can define this to use the generic x86 watchpoint support; see Chapter 3 [Algorithms], page 2.

SYMBOLS_CAN_START_WITH_DOLLAR

> Some systems have routines whose names start with '$'. Giving this macro a non-zero value tells GDB's expression parser to check for such routines when parsing tokens that begin with '$'.
>
> On HP-UX, certain system routines (millicode) have names beginning with '$' or '$$'. For example, $$dyncall is a millicode routine that handles inter-space procedure calls on PA-RISC.

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.

gdbarch_in_function_epilogue_p (*gdbarch*, *pc*)

> Returns non-zero if the given *pc* is in the epilogue of a function. The epilogue of a function is defined as the part of a function where the stack frame of the function already has been destroyed up to the final 'return from function call' instruction.

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.

IN_SOLIB_DYNSYM_RESOLVE_CODE (*pc*)

>   Define this to evaluate to nonzero if the program is stopped in the dynamic
>   linker.

SKIP_SOLIB_RESOLVER (*pc*)

>   Define this to evaluate to the (nonzero) address at which execution should
>   continue to get past the dynamic linker's symbol resolution function. A zero
>   value indicates that it is not important or necessary to set a breakpoint to get
>   through the dynamic linker and that single stepping will suffice.

INTEGER_TO_ADDRESS (*type*, *buf*)

>   Define this when the architecture needs to handle non-pointer to address con-
>   versions specially. Converts that value to an address according to the current
>   architectures conventions.
>
>   *Pragmatics: When the user copies a well defined expression from their source
>   code and passes it, as a parameter, to GDB's* `print` *command, they should
>   get the same value as would have been computed by the target program. Any
>   deviation from this rule can cause major confusion and annoyance, and needs
>   to be justified carefully. In other words, GDB doesn't really have the freedom to
>   do these conversions in clever and useful ways. It has, however, been pointed
>   out that users aren't complaining about how GDB casts integers to pointers;
>   they are complaining that they can't take an address from a disassembly listing
>   and give it to* `x/i`*. Adding an architecture method like* INTEGER_TO_ADDRESS
>   *certainly makes it possible for GDB to "get it right" in all circumstances.*
>
>   See Chapter 9 [Pointers Are Not Always Addresses], page 33.

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. Cur-
>   rently 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.)

POINTER_TO_ADDRESS (*type*, *buf*)

> Assume that *buf* holds a pointer of type *type*, in the appropriate format for the current architecture. Return the byte address the pointer refers to. See Chapter 9 [Pointers Are Not Always Addresses], page 33.

REGISTER_CONVERTIBLE (*reg*)

> Return non-zero if *reg* uses different raw and virtual formats. See Chapter 9 [Raw and Virtual Register Representations], page 33.

REGISTER_TO_VALUE(*regnum*, *type*, *from*, *to*)

> Convert the raw contents of register *regnum* into a value of type *type*. See Chapter 9 [Using Different Register and Memory Data Representations], page 33.

REGISTER_RAW_SIZE (*reg*)

> Return the raw size of *reg*; defaults to the size of the register's virtual type. See Chapter 9 [Raw and Virtual Register Representations], page 33.

REGISTER_VIRTUAL_SIZE (*reg*)

> Return the virtual size of *reg*; defaults to the size of the register's virtual type. Return the virtual size of *reg*. See Chapter 9 [Raw and Virtual Register Representations], page 33.

REGISTER_VIRTUAL_TYPE (*reg*)

> Return the virtual type of *reg*. See Chapter 9 [Raw and Virtual Register Representations], page 33.

REGISTER_CONVERT_TO_VIRTUAL(*reg*, *type*, *from*, *to*)

> Convert the value of register *reg* from its raw form to its virtual form. See Chapter 9 [Raw and Virtual Register Representations], page 33.

REGISTER_CONVERT_TO_RAW(*type*, *reg*, *from*, *to*)

> Convert the value of register *reg* from its virtual form to its raw form. See Chapter 9 [Raw and Virtual Register Representations], page 33.

RETURN_VALUE_ON_STACK(*type*)

> Return non-zero if values of type TYPE are returned on the stack, using the "struct convention" (i.e., the caller provides a pointer to a buffer in which the callee should store the return value). This controls how the 'finish' command finds a function's return value, and whether an inferior function call reserves space on the stack for the return value.
>
> The full logic GDB uses here is kind of odd.
>
> - If the type being returned by value is not a structure, union, or array, and RETURN_VALUE_ON_STACK returns zero, then GDB concludes the value is not returned using the struct convention.
> - Otherwise, GDB calls USE_STRUCT_CONVENTION (see below). If that returns non-zero, GDB assumes the struct convention is in use.
>
> In other words, to indicate that a given type is returned by value using the struct convention, that type must be either a struct, union, array, or something RETURN_VALUE_ON_STACK likes, *and* something that USE_STRUCT_CONVENTION likes.

Note that, in C and C++, arrays are never returned by value. In those languages, these predicates will always see a pointer type, never an array type. All the references above to arrays being returned by value apply only to other languages.

`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 (depending on *insert_breapoints_p*) breakpoints at each possible destinations of the next instruction. See '`sparc-tdep.c`' and '`rs6000-tdep.c`' for examples.

`SOFUN_ADDRESS_MAYBE_MISSING`

Somebody clever observed that, the more actual addresses you have in the debug information, the more time the linker has to spend relocating them. So whenever there's some other way the debugger could find the address it needs, you should omit it from the debug info, to make linking faster.

`SOFUN_ADDRESS_MAYBE_MISSING` indicates that a particular set of hacks of this sort are in use, affecting `N_SO` and `N_FUN` entries in stabs-format debugging information. `N_SO` stabs mark the beginning and ending addresses of compilation units in the text segment. `N_FUN` stabs mark the starts and ends of functions.

`SOFUN_ADDRESS_MAYBE_MISSING` means two things:

- `N_FUN` stabs have an address of zero. Instead, you should find the addresses where the function starts by taking the function name from the stab, and then looking that up in the minsyms (the linker/assembler symbol table). In other words, the stab has the name, and the linker/assembler symbol table is the only place that carries the address.

- `N_SO` stabs have an address of zero, too. You just look at the `N_FUN` stabs that appear before and after the `N_SO` stab, and guess the starting and ending addresses of the compilation unit from them.

`PCC_SOL_BROKEN`

(Used only in the Convex target.)

`PC_IN_CALL_DUMMY`

See '`inferior.h`'.

`PC_IN_SIGTRAMP (`*pc*`, `*name*`)`

The *sigtramp* is a routine that the kernel calls (which then calls the signal handler). On most machines it is a library routine that is linked into the executable.

This function, given a program counter value in *pc* and the (possibly NULL) name of the function in which that *pc* resides, returns nonzero if the *pc* and/or *name* show that we are in sigtramp.

`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.

PARM_BOUNDARY

> If non-zero, round arguments to a boundary of this many bits before pushing them on the stack.

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 and in 'return_command' to remove a real stack frame.

PUSH_ARGUMENTS (*nargs*, *args*, *sp*, *struct_return*, *struct_addr*)

> Define this to push arguments onto the stack for inferior function call. Returns 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.

SKIP_PERMANENT_BREAKPOINT

> Advance the inferior's PC past a permanent breakpoint. GDB normally steps over a breakpoint by removing it, stepping one instruction, and re-inserting the breakpoint. However, permanent breakpoints are hardwired into the inferior, and can't be removed, so this strategy doesn't work. Calling `SKIP_PERMANENT_BREAKPOINT` adjusts the processor's state so that execution will resume just after the breakpoint. This macro does the right thing even when the breakpoint is in the delay slot of a branch or jump.

SKIP_PROLOGUE (*pc*)

> A C expression that returns the address of the "real" code beyond the function entry prologue found at *pc*.

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, regcache, valbuf*)

> A C expression that writes the function return value, found in *valbuf*, into the *regcache*. *type* is the type of the value that is to be returned.

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_CHAR_BIT
>   Number of bits in a char; defaults to 8.

TARGET_CHAR_SIGNED
>   Non-zero if `char` is normally signed on this architecture; zero if it should be
>   unsigned.
>
>   The ISO C standard requires the compiler to treat `char` as equivalent to either
>   `signed char` or `unsigned char`; any character in the standard execution set
>   is supposed to be positive. Most compilers treat `char` as signed, but `char` is
>   unsigned on the IBM S/390, RS6000, and PowerPC targets.

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`

> These change the behavior of `read_pc`, `write_pc`, `read_sp`, `write_sp` and `read_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.

`TARGET_HAS_HARDWARE_WATCHPOINTS`

> If non-zero, the target has support for hardware-assisted watchpoints. See Chapter 3 [Algorithms], page 2, for more details and other related macros.

`TARGET_PRINT_INSN (`*addr*`, `*info*`)`

> This is the function used by GDB to print an assembly instruction. It prints the instruction at address *addr* in debugged memory and returns the length of the instruction, in bytes. If a target doesn't define its own printing routine, it defaults to an accessor function for the global pointer `tm_print_insn`. This usually points to a function in the `opcodes` library (see Chapter 12 [Opcodes], page 63). *info* is a structure (of type `disassemble_info`) defined in 'include/dis-asm.h' used to pass information to the instruction decoding routine.

`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.

`VALUE_TO_REGISTER(`*type*`, `*regnum*`, `*from*`, `*to*`)`

> Convert a value of type *type* into the raw contents of register *regnum*'s. See Chapter 9 [Using Different Register and Memory Data Representations], page 33.

`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`.

9.10 Adding a New Target

The following files add 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=...' and 'TDEPLIBS=...'. 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, replaced by autoconf, and may go away in future versions of GDB.

'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/*arch*-tdep.c'
'gdb/*arch*-tdep.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 '*ttt*-tdep.h'. It can be shared among many targets that use the same processor.

'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.
>
> New targets do not need this file and should not create it.

'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.
>
> New targets do not need this file and should not create it.

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'.

9.11 Converting an existing Target Architecture to Multi-arch

This section describes the current accepted best practice for converting an existing target architecture to the multi-arch framework.

The process consists of generating, testing, posting and committing a sequence of patches. Each patch must contain a single change, for instance:

- Directly convert a group of functions into macros (the conversion does not change the behavior of any of the functions).
- Replace a non-multi-arch with a multi-arch mechanism (e.g., `FRAME_INFO`).
- Enable multi-arch level one.
- Delete one or more files.

There isn't a size limit on a patch, however, a developer is strongly encouraged to keep the patch size down.

Since each patch is well defined, and since each change has been tested and shows no regressions, the patches are considered *fairly* obvious. Such patches, when submitted by developers listed in the '`MAINTAINERS`' file, do not need approval. Occasional steps in the process may be more complicated and less clear. The developer is expected to use their judgment and is encouraged to seek advice as needed.

9.11.1 Preparation

The first step is to establish control. Build (with '`-Werror`' enabled) and test the target so that there is a baseline against which the debugger can be compared.

At no stage can the test results regress or GDB stop compiling with '`-Werror`'.

9.11.2 Add the multi-arch initialization code

The objective of this step is to establish the basic multi-arch framework. It involves

- The addition of a *arch*`_gdbarch_init` function[4] that creates the architecture:

```
static struct gdbarch *
d10v_gdbarch_init (info, arches)
     struct gdbarch_info info;
     struct gdbarch_list *arches;
{
  struct gdbarch *gdbarch;
  /* there is only one d10v architecture */
  if (arches != NULL)
    return arches->gdbarch;
  gdbarch = gdbarch_alloc (&info, NULL);
  return gdbarch;
}
```

- A per-architecture dump function to print any architecture specific information:

---

[4] The above is from the original example and uses K&R C. GDB has since converted to ISO C but lets ignore that.

```
    static void
    mips_dump_tdep (struct gdbarch *current_gdbarch,
                    struct ui_file *file)
    {
        ... code to print architecture specific info ...
    }
```

- A change to _initialize_*arch*_tdep to register this new architecture:

```
    void
    _initialize_mips_tdep (void)
    {
        gdbarch_register (bfd_arch_mips, mips_gdbarch_init,
                          mips_dump_tdep);
    }
```

- Add the macro `GDB_MULTI_ARCH`, defined as 0 (zero), to the file '`config/`*arch*`/tm-`*arch*`.h`'.

9.11.3 Update multi-arch incompatible mechanisms

Some mechanisms do not work with multi-arch. They include:

`EXTRA_FRAME_INFO`
> Delete.

`FRAME_FIND_SAVED_REGS`
> Replaced with `FRAME_INIT_SAVED_REGS`

At this stage you could also consider converting the macros into functions.

9.11.4 Prepare for multi-arch level to one

Temporally set `GDB_MULTI_ARCH` to `GDB_MULTI_ARCH_PARTIAL` and then build and start GDB (the change should not be committed). GDB may not build, and once built, it may die with an internal error listing the architecture methods that must be provided.

Fix any build problems (patch(es)).

Convert all the architecture methods listed, which are only macros, into functions (patch(es)).

Update *arch*_`gdbarch_init` to set all the missing architecture methods and wrap the corresponding macros in `#if !GDB_MULTI_ARCH` (patch(es)).

9.11.5 Set multi-arch level one

Change the value of `GDB_MULTI_ARCH` to GDB_MULTI_ARCH_PARTIAL (a single patch).

Any problems with throwing "the switch" should have been fixed already.

9.11.6 Convert remaining macros

Suggest converting macros into functions (and setting the corresponding architecture method) in small batches.

9.11.7 Set multi-arch level to two

This should go smoothly.

9.11.8 Delete the TM file

The '`tm-`*arch*`.h`' can be deleted. '*arch*`.mt`' and '`configure.in`' updated.

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 set up 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 by a *native* configuration 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'.

> *Maintainer's note: The '.mh' suffix is because this file originally contained 'Makefile' fragments for hosting GDB on machine xyz. While the file is no longer used for this purpose, the '.mh' suffix remains. Perhaps someone will eventually rename these fragments so that they have a '.mn' suffix.*

'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.

I386_USE_GENERIC_WATCHPOINTS

An x86-based machine can define this to use the generic x86 watchpoint support; see Chapter 3 [Algorithms], page 2.

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.

PREPARE_TO_PROCEED (*select_it*)

This (ugly) macro allows a native configuration to customize the way the `proceed` function in 'infrun.c' deals with switching between threads.

In a multi-threaded task we may select another thread and then continue or step. But if the old thread was stopped at a breakpoint, it will immediately cause another breakpoint stop without any execution (i.e. it will report a breakpoint hit incorrectly). So GDB must step over it first.

If defined, `PREPARE_TO_PROCEED` should check the current thread against the thread that reported the most recent event. If a step-over is required, it returns TRUE. If *select_it* is non-zero, it should reselect the old thread.

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

See '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*, *readsyms*)
> Define this to expand into an expression that will cause the symbols in *filename*
> to be added to GDB's symbol table. If *readsyms* is zero symbols are not read
> but any necessary low level processing for *filename* is still done.

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
> See '`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 (e.g., `xmalloc` some memory, or `open` a file) that needs to be undone later (e.g., `xfree` 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 and control returns to the top level; when an error occurs and the stack is unwound; or when your code decides it's time to explicitly perform cleanups. Alternatively you can elect to discard the cleanups you created.

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 later be passed to `do_cleanups` or `discard_cleanups`. Unless you are going to call `do_cleanups` or `discard_cleanups`, you can ignore the result from `make_cleanup`.

`do_cleanups (`*old_chain*`);`

> Do all cleanups added to the chain since the corresponding `make_cleanup` call was made.

`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.

Cleanups are implemented as a chain. The handle returned by `make_cleanups` includes the cleanup passed to the call and any later cleanups appended to the chain (but not yet discarded or performed). E.g.:

```
make_cleanup (a, 0);
{
  struct cleanup *old = make_cleanup (b, 0);
  make_cleanup (c, 0)
  ...
  do_cleanups (old);
}
```

will call `c()` and `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.

Your function should explicitly do or discard the cleanups it creates. Failing to do this leads to non-deterministic behavior since the caller will arbitrarily do or discard your functions cleanups. This need leads to two common cleanup styles.

The first style is try/finally. Before it exits, your code-block calls `do_cleanups` with the old cleanup chain and thus ensures that your code-block's cleanups are always performed. For instance, the following code-segment avoids a memory leak problem (even when `error` is called and a forced stack unwind occurs) by ensuring that the `xfree` will always be called:

```
struct cleanup *old = make_cleanup (null_cleanup, 0);
data = xmalloc (sizeof blah);
make_cleanup (xfree, data);
... blah blah ...
do_cleanups (old);
```

The second style is try/except. Before it exits, your code-block calls `discard_cleanups` with the old cleanup chain and thus ensures that any created cleanups are not performed. For instance, the following code segment, ensures that the file will be closed but only if there is an error:

```
FILE *file = fopen ("afile", "r");
struct cleanup *old = make_cleanup (close_file, file);
... blah blah ...
discard_cleanups (old);
return file;
```

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 Per-architecture module data

The multi-arch framework includes a mechanism for adding module specific per-architecture data-pointers to the `struct gdbarch` architecture object.

A module registers one or more per-architecture data-pointers using the function `register_gdbarch_data`:

struct gdbarch_data *`*register_gdbarch_data`   *Function*
        (gdbarch_data_init_ftype *`init`, gdbarch_data_free_ftype *`free`)
    The *init* function is used to obtain an initial value for a per-architecture data-pointer. The function is called, after the architecture has been created, when the data-pointer is still uninitialized (`NULL`) and its value has been requested via a call to `gdbarch_data`. A data-pointer can also be initialize explicitly using `set_gdbarch_data`.

    The *free* function is called when a data-pointer needs to be destroyed. This occurs when either the corresponding `struct gdbarch` object is being destroyed or when `set_gdbarch_data` is overriding a non-NULL data-pointer value.

    The function `register_gdbarch_data` returns a `struct gdbarch_data` that is used to identify the data-pointer that was added to the module.

A typical module has `init` and `free` functions of the form:

```
static struct gdbarch_data *nozel_handle;
static void *
nozel_init (struct gdbarch *gdbarch)
{
  struct nozel *data = XMALLOC (struct nozel);
  ...
  return data;
}
...
static void
nozel_free (struct gdbarch *gdbarch, void *data)
{
  xfree (data);
}
```

Since uninitialized (`NULL`) data-pointers are initialized on-demand, an `init` function is free to call other modules that use data-pointers. Those modules data-pointers will be initialized as needed. Care should be taken to ensure that the `init` call graph does not contain cycles.

The data-pointer is registered with the call:

```
void
_initialize_nozel (void)
{
  nozel_handle = register_gdbarch_data (nozel_init, nozel_free);
...
```

The per-architecture data-pointer is accessed using the function:

void ***gdbarch_data** (struct gdbarch *_gdbarch_, struct   Function
        gdbarch_data *_data_handle_)

   Given the architecture _arch_ and module data handle _data_handle_ (returned
   by `register_gdbarch_data`, this function returns the current value of the
   per-architecture data-pointer.

The non-`NULL` data-pointer returned by `gdbarch_data` should be saved in a local variable and then used directly:

```
int
nozel_total (struct gdbarch *gdbarch)
{
  int total;
  struct nozel *data = gdbarch_data (gdbarch, nozel_handle);
  ...
  return total;
}
```

It is also possible to directly initialize the data-pointer using:

void **set_gdbarch_data** (struct gdbarch *gdbarch, struct   *Function*
      gdbarch_data *handle, void *pointer*)

    Update the data-pointer corresponding to *handle* with the value of *pointer*. If the
    previous data-pointer value is non-NULL, then it is freed using data-pointers *free*
    function.

This function is used by modules that require a mechanism for explicitly setting the
per-architecture data-pointer during architecture creation:

```
/* Called during architecture creation.  */
extern void
set_gdbarch_nozel (struct gdbarch *gdbarch,
                   int total)
{
  struct nozel *data = XMALLOC (struct nozel);
  ...
  set_gdbarch_data (gdbarch, nozel_handle, nozel);
}

/* Default, called when nozel not set by set_gdbarch_nozel().  */
static void *
nozel_init (struct gdbarch *gdbarch)
{
  struct nozel *default_nozel = XMALLOC (struc nozel);
  ...
  return default_nozel;
}

void
_initialize_nozel (void)
{
  nozel_handle = register_gdbarch_data (nozel_init, NULL);
  ...
```

Note that an `init` function still needs to be registered. It is used to initialize the data-
pointer when the architecture creation phase fail to set an initial value.

13.3 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.

13.4 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.

13.4.1 ISO C

GDB assumes an ISO/IEC 9899:1990 (a.k.a. ISO C90) compliant compiler.

GDB does not assume an ISO C or POSIX compliant C library.

13.4.2 Memory Management

GDB does not use the functions `malloc`, `realloc`, `calloc`, `free` and `asprintf`.

GDB uses the functions `xmalloc`, `xrealloc` and `xcalloc` when allocating memory. Unlike `malloc` et.al. these functions do not return when the memory pool is empty. Instead, they unwind the stack using cleanups. These functions return `NULL` when requested to allocate a chunk of memory of size zero.

Pragmatics: By using these functions, the need to check every memory allocation is removed. These functions provide portable behavior.

GDB does not use the function `free`.

GDB uses the function `xfree` to return memory to the memory pool. Consistent with ISO-C, this function ignores a request to free a `NULL` pointer.

Pragmatics: On some systems `free` *fails when passed a* `NULL` *pointer.*

GDB can use the non-portable function `alloca` for the allocation of small temporary values (such as strings).

Pragmatics: This function is very non-portable. Some systems restrict the memory being allocated to no more than a few kilobytes.

GDB uses the string function `xstrdup` and the print function `xasprintf`.

Pragmatics: `asprintf` *and* `strdup` *can fail. Print functions such as* `sprintf` *are very prone to buffer overflow errors.*

13.4.3 Compiler Warnings

With few exceptions, developers should include the configuration option '`--enable-gdb-build-warnings=,-Werror`' when building GDB. The exceptions are listed in the file '`gdb/MAINTAINERS`'.

This option causes GDB (when built using GCC) to be compiled with a carefully selected list of compiler warning flags. Any warnings from those flags being treated as errors.

The current list of warning flags includes:

'-Wimplicit'
> Since GDB coding standard requires all functions to be declared using a prototype, the flag has the side effect of ensuring that prototyped functions are always visible with out resorting to '-Wstrict-prototypes'.

'-Wreturn-type'
> Such code often appears to work except on instruction set architectures that use register windows.

'-Wcomment'
'-Wtrigraphs'
'-Wformat'
> Since GDB uses the `format printf` attribute on all `printf` like functions this checks not just `printf` calls but also calls to functions such as `fprintf_unfiltered`.

'-Wparentheses'
> This warning includes uses of the assignment operator within an `if` statement.

'-Wpointer-arith'
'-Wuninitialized'

Pragmatics: Due to the way that GDB is implemented most functions have unused parameters. Consequently the warning '-Wunused-parameter' is precluded from the list. The macro `ATTRIBUTE_UNUSED` is not used as it leads to false negatives — it is not an error to have `ATTRIBUTE_UNUSED` on a parameter that is being used. The options '-Wall' and '-Wunused' are also precluded because they both include '-Wunused-parameter'.

Pragmatics: GDB has not simply accepted the warnings enabled by '-Wall -Werror -W...'. Instead it is selecting warnings when and where their benefits can be demonstrated.

13.4.4 Formatting

The standard GNU recommendations for formatting must be followed strictly.

A function declaration should not have its name in column zero. A function definition should have its name in column zero.

```
/* Declaration */
static void foo (void);
/* Definition */
void
foo (void)
{
}
```

Pragmatics: This simplifies scripting. Function definitions can be found using '^function-name'.

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` utilities.

Pointers are declared using the traditional K&R C style:

```
void *foo;
```

and not:

```
void * foo;
void* foo;
```

13.4.5 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.

13.4.6 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.

Macros are bad, M'kay. (But if you have to use a macro, make sure that the macro arguments are protected with parentheses.)

Declarations like '`struct foo *`' should be used in preference to declarations like '`typedef struct foo { ... } *foo_ptr`'.

13.4.7 Function Prototypes

Prototypes must be used when both *declaring* and *defining* a function. Prototypes for GDB functions must include both the argument type and name, with the name matching that used in the actual function definition.

All external functions should have a 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.

Where a source file needs a forward declaration of a static function, that declaration must appear in a block near the top of the source file.

13.4.8 Internal Error Recovery

During its execution, GDB can encounter two types of errors. User errors and internal errors. User errors include not only a user entering an incorrect command but also problems arising from corrupt object files and system errors when interacting with the target. Internal errors include situations where GDB has detected, at run time, a corrupt or erroneous situation.

When reporting an internal error, GDB uses `internal_error` and `gdb_assert`.

GDB must not call `abort` or `assert`.

Pragmatics: There is no `internal_warning` *function. Either the code detected a user error, recovered from it and issued a* `warning` *or the code failed to correctly recover from the user error and issued an* `internal_error`.

13.4.9 File Names

Any file used when building the core of GDB must be in lower case. Any file used when building the core of GDB must be 8.3 unique. These requirements apply to both source and generated files.

Pragmatics: The core of GDB must be buildable on many platforms including DJGPP and MacOS/HFS. Every time an unfriendly file is introduced to the build process both 'Makefile.in' *and* 'configure.in' *need to be modified accordingly. Compare the convoluted conversion process needed to transform* 'COPYING' *into* 'copying.c' *with the conversion needed to transform* 'version.in' *into* 'version.c'.

Any file non 8.3 compliant file (that is not used when building the core of GDB) must be added to 'gdb/config/djgpp/fnchange.lst'.

Pragmatics: This is clearly a compromise.

When GDB has a local version of a system header file (ex 'string.h') the file name based on the POSIX header prefixed with 'gdb_' ('gdb_string.h').

For other files '-' is used as the separator.

13.4.10 Include Files

A '.c' file should include 'defs.h' first.

A '.c' file should directly include the `.h` file of every declaration and/or definition it directly refers to. It cannot rely on indirect inclusion.

A '`.h`' file should directly include the `.h` file of every declaration and/or definition it directly refers to. It cannot rely on indirect inclusion. Exception: The file '`defs.h`' does not need to be directly included.

An external declaration should only appear in one include file.

An external declaration should never appear in a `.c` file. Exception: a declaration for the `_initialize` function that pacifies '`-Wmissing-declaration`'.

A `typedef` definition should only appear in one include file.

An opaque `struct` declaration can appear in multiple '`.h`' files. Where possible, a '`.h`' file should use an opaque `struct` declaration instead of an include.

All '`.h`' files should be wrapped in:

```
#ifndef INCLUDE_FILE_NAME_H
#define INCLUDE_FILE_NAME_H
header body
#endif
```

13.4.11 Clean Design and Portable Implementation

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.

One particularly notorious area where system dependencies tend to creep in is handling of file names. The mainline GDB code assumes Posix semantics of file names: absolute file names begin with a forward slash '`/`', slashes are used to separate leading directories,

case-sensitive file names. These assumptions are not necessarily true on non-Posix systems
such as MS-Windows. To avoid system-dependent code where you need to take apart or
construct a file name, use the following portable macros:

HAVE_DOS_BASED_FILE_SYSTEM
> This preprocessing symbol is defined to a non-zero value on hosts whose filesys-
> tems belong to the MS-DOS/MS-Windows family. Use this symbol to write
> conditional code which should only be compiled for such hosts.

IS_DIR_SEPARATOR (*c*)
> Evaluates to a non-zero value if *c* is a directory separator character. On Unix
> and GNU/Linux systems, only a slash '/' is such a character, but on Windows,
> both '/' and '\' will pass.

IS_ABSOLUTE_PATH (*file*)
> Evaluates to a non-zero value if *file* is an absolute file name. For Unix and
> GNU/Linux hosts, a name which begins with a slash '/' is absolute. On DOS
> and Windows, 'd:/foo' and 'x:\bar' are also absolute file names.

FILENAME_CMP (*f1*, *f2*)
> Calls a function which compares file names *f1* and *f2* as appropriate for the
> underlying host filesystem. For Posix systems, this simply calls strcmp; on
> case-insensitive filesystems it will call strcasecmp instead.

DIRNAME_SEPARATOR
> Evaluates to a character which separates directories in PATH-style lists, typically
> held in environment variables. This character is ':' on Unix, ';' on DOS and
> Windows.

SLASH_STRING
> This evaluates to a constant string you should use to produce an absolute
> filename from leading directories and the file's basename. SLASH_STRING is "/"
> on most systems, but might be "\\" for some Windows-based ports.

In addition to using these macros, be sure to use portable library functions whenever
possible. For example, to extract a directory or a basename part from a file name, use the
dirname and basename library functions (available in libiberty for platforms which don't
provide them), instead of searching for a slash with strrchr.

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 #ifdefs 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.

All debugging code must be controllable using the '`set debug` *module*' command. Do
not use `printf` to print trace messages. Use `fprintf_unfiltered(gdb_stdlog, ...`. Do
not use `#ifdef DEBUG`.

14 Porting GDB

Most of the work in making GDB compile on a new machine is in specifying the config-
uration 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).

  *Maintainer's note: Work in progress. The file '`gdb/configure.host`' originally needed
  to be modified when either a new native target or a new host machine was being added
  to GDB. Recent changes have removed this requirement. The file now only needs to be
  modified when adding a new native configuration. This will likely changed again in the
  future.*

- 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 Releasing GDB

15.1 Versions and Branches

15.1.1 Version Identifiers

GDB's version is determined by the file '`gdb/version.in`'.

GDB's mainline uses ISO dates to differentiate between versions. The CVS repository uses *YYYY-MM-DD*-cvs while the corresponding snapshot uses *YYYYMMDD*.

GDB's release branch uses a slightly more complicated scheme. When the branch is first cut, the mainline version identifier is prefixed with the *major.minor* from of the previous release series but with .90 appended. As draft releases are drawn from the branch, the minor minor number (.90) is incremented. Once the first release (*M.N*) has been made, the version prefix is updated to *M.N*.0.90 (dot zero, dot ninety). Follow on releases have an incremented minor minor version number (.0).

Using 5.1 (previous) and 5.2 (current), the example below illustrates a typical sequence of version identifiers:

5.1.1   final release from previous branch

2002-03-03-cvs

    main-line the day the branch is cut

5.1.90-2002-03-03-cvs

    corresponding branch version

5.1.91   first draft release candidate

5.1.91-2002-03-17-cvs
              updated branch version

5.1.92   second draft release candidate

5.1.92-2002-03-31-cvs
              updated branch version

5.1.93   final release candidate (see below)

5.2   official release

5.2.0.90-2002-04-07-cvs
              updated CVS branch version

5.2.1   second official release

Notes:

- Minor minor minor draft release candidates such as 5.2.0.91 have been omitted from the example. Such release candidates are, typically, never made.

- For 5.1.93 the bziped tar ball 'gdb-5.1.93.tar.bz2' is just the official 'gdb-5.2.tar' renamed and compressed.

To avoid version conflicts, vendors are expected to modify the file 'gdb/version.in' to include a vendor unique alphabetic identifier (an official GDB release never uses alphabetic characters in its version identifer).

Since GDB does not make minor minor minor releases (e.g., 5.1.0.1) the conflict between that and a minor minor draft release identifier (e.g., 5.1.0.90) is avoided.

15.1.2 Branches

GDB draws a release series (5.2, 5.2.1, . . .) from a single release branch (gdb_5_2-branch). Since minor minor minor releases (5.1.0.1) are not made, the need to branch the release branch is avoided (it also turns out that the effort required for such a a branch and release is significantly greater than the effort needed to create a new release from the head of the release branch).

Releases 5.0 and 5.1 used branch and release tags of the form:

```
gdb_N_M-YYYY-MM-DD-branchpoint
gdb_N_M-YYYY-MM-DD-branch
gdb_M_N-YYYY-MM-DD-release
```

Release 5.2 is trialing the branch and release tags:

```
gdb_N_M-YYYY-MM-DD-branchpoint
gdb_N_M-branch
gdb_M_N-YYYY-MM-DD-release
```

Pragmatics: The branchpoint and release tags need to identify when a branch and release are made. The branch tag, denoting the head of the branch, does not have this criteria.

15.2 Branch Commit Policy

The branch commit policy is pretty slack. GDB releases 5.0, 5.1 and 5.2 all used the below:

- The 'gdb/MAINTAINERS' file still holds.
- Don't fix something on the branch unless/until it is also fixed in the trunk. If this isn't possible, mentioning it in the 'gdb/PROBLEMS' file is better than committing a hack.
- When considering a patch for the branch, suggested criteria include: Does it fix a build? Does it fix the sequence *break main; run* when debugging a static binary?
- The further a change is from the core of GDB, the less likely the change will worry anyone (e.g., target specific code).
- Only post a proposal to change the core of GDB after you've sent individual bribes to all the people listed in the 'MAINTAINERS' file ;-)

Pragmatics: Provided updates are restricted to non-core functionality there is little chance that a broken change will be fatal. This means that changes such as adding a new architectures or (within reason) support for a new host are considered acceptable.

15.3 Obsoleting code

Before anything else, poke the other developers (and around the source code) to see if there is anything that can be removed from GDB (an old target, an unused file).

Obsolete code is identified by adding an OBSOLETE prefix to every line. Doing this means that it is easy to identify something that has been obsoleted when greping through the sources.

The process is done in stages — this is mainly to ensure that the wider GDB community has a reasonable opportunity to respond. Remember, everything on the Internet takes a week.

1. Post the proposal on the GDB mailing list Creating a bug report to track the task's state, is also highly recommended.
2. Wait a week or so.
3. Post the proposal on the GDB Announcement mailing list.
4. Wait a week or so.
5. Go through and edit all relevant files and lines so that they are prefixed with the word OBSOLETE.
6. Wait until the next GDB version, containing this obsolete code, has been released.
7. Remove the obsolete code.

Maintainer note: While removing old code is regrettable it is hopefully better for GDB's long term development. Firstly it helps the developers by removing code that is either no longer relevant or simply wrong. Secondly since it removes any history associated with the file (effectively clearing the slate) the developer has a much freer hand when it comes to fixing broken files.

15.4 Before the Branch

The most important objective at this stage is to find and fix simple changes that become a pain to track once the branch is created. For instance, configuration problems that stop GDB from even building. If you can't get the problem fixed, document it in the 'gdb/PROBLEMS' file.

Prompt for 'gdb/NEWS'

People always forget. Send a post reminding them but also if you know something interesting happened add it yourself. The `schedule` script will mention this in its e-mail.

Review 'gdb/README'

Grab one of the nightly snapshots and then walk through the 'gdb/README' looking for anything that can be improved. The `schedule` script will mention this in its e-mail.

Refresh any imported files.

A number of files are taken from external repositories. They include:

- 'texinfo/texinfo.tex'
- 'config.guess' et. al. (see the top-level 'MAINTAINERS' file)
- 'etc/standards.texi', 'etc/make-stds.texi'

Check the ARI

A.R.I. is an `awk` script (Awk Regression Index ;-) that checks for a number of errors and coding conventions. The checks include things like using `malloc` instead of `xmalloc` and file naming problems. There shouldn't be any regressions.

15.4.1 Review the bug data base

Close anything obviously fixed.

15.4.2 Check all cross targets build

The targets are listed in 'gdb/MAINTAINERS'.

15.5 Cut the Branch

Create the branch

```
$  u=5.1
$  v=5.2
$  V=`echo $v | sed 's/\./_/g'`
$  D=`date -u +%Y-%m-%d`
$  echo $u $V $D
5.1 5_2 2002-03-03
```

```
$  echo cvs -f -d :ext:sources.redhat.com:/cvs/src rtag \
-D $D-gmt gdb_$V-$D-branchpoint insight+dejagnu
cvs -f -d :ext:sources.redhat.com:/cvs/src rtag
-D 2002-03-03-gmt gdb_5_2-2002-03-03-branchpoint insight+dejagnu
$  ^echo ^^
...
$  echo cvs -f -d :ext:sources.redhat.com:/cvs/src rtag \
-b -r gdb_$V-$D-branchpoint gdb_$V-branch insight+dejagnu
cvs -f -d :ext:sources.redhat.com:/cvs/src rtag \
-b -r gdb_5_2-2002-03-03-branchpoint gdb_5_2-branch insight+dejagnu
$  ^echo ^^
...
$
```

- by using `-D YYYY-MM-DD-gmt` the branch is forced to an exact date/time.
- the trunk is first taged so that the branch point can easily be found
- Insight (which includes GDB) and dejagnu are all tagged at the same time
- '`version.in`' gets bumped to avoid version number conflicts
- the reading of '`.cvsrc`' is disabled using '`-f`'

Update '`version.in`'

```
$  u=5.1
$  v=5.2
$  V=`echo $v | sed 's/\./_/g'`
$  echo $u $v$V
5.1 5_2
$  cd /tmp
$  echo cvs -f -d :ext:sources.redhat.com:/cvs/src co \
-r gdb_$V-branch src/gdb/version.in
cvs -f -d :ext:sources.redhat.com:/cvs/src co
 -r gdb_5_2-branch src/gdb/version.in
$  ^echo ^^
U src/gdb/version.in
$  cd src/gdb
$  echo $u.90-0000-00-00-cvs > version.in
$  cat version.in
5.1.90-0000-00-00-cvs
$  cvs -f commit version.in
```

- '`0000-00-00`' is used as a date to pump prime the version.in update mechanism
- '`.90`' and the previous branch version are used as fairly arbitrary initial branch version number

Update the web and news pages

Something?

Tweak cron to track the new branch

The file 'gdbadmin/cron/crontab' contains gdbadmin's cron table. This file needs to be updated so that:

- a daily timestamp is added to the file 'version.in'
- the new branch is included in the snapshot process

See the file 'gdbadmin/cron/README' for how to install the updated cron table.

The file 'gdbadmin/ss/README' should also be reviewed to reflect any changes. That file is copied to both the branch/ and current/ snapshot directories.

Update the NEWS and README files

The 'NEWS' file needs to be updated so that on the branch it refers to *changes in the current release* while on the trunk it also refers to *changes since the current release*.

The 'README' file needs to be updated so that it refers to the current release.

Post the branch info

Send an announcement to the mailing lists:

- GDB Announcement mailing list
- GDB Discsussion mailing list and GDB Discsussion mailing list

Pragmatics: The branch creation is sent to the announce list to ensure that people people not subscribed to the higher volume discussion list are alerted.

The announcement should include:

- the branch tag
- how to check out the branch using CVS
- the date/number of weeks until the release
- the branch commit policy still holds.

15.6 Stabilize the branch

Something goes here.

15.7 Create a Release

The process of creating and then making available a release is broken down into a number of stages. The first part addresses the technical process of creating a releasable tar ball. The later stages address the process of releasing that tar ball.

When making a release candidate just the first section is needed.

15.7.1 Create a release candidate

The objective at this stage is to create a set of tar balls that can be made available as a formal release (or as a less formal release candidate).

Freeze the branch

Send out an e-mail notifying everyone that the branch is frozen to gdb-patches@sources.redhat.com.

Establish a few defaults.

```
$  b=gdb_5_2-branch
$  v=5.2
$  t=/sourceware/snapshot-tmp/gdbadmin-tmp
$  echo $t/$b/$v
/sourceware/snapshot-tmp/gdbadmin-tmp/gdb_5_2-branch/5.2
$  mkdir -p $t/$b/$v
$  cd $t/$b/$v
$  pwd
/sourceware/snapshot-tmp/gdbadmin-tmp/gdb_5_2-branch/5.2
$  which autoconf
/home/gdbadmin/bin/autoconf
$
```

Notes:

- Check the `autoconf` version carefully. You want to be using the version taken from the 'binutils' snapshot directory, which can be found at ftp://sources.redhat.com/pub/binutils/. It is very unlikely that a system installed version of autoconf (e.g., '/usr/bin/autoconf') is correct.

Check out the relevant modules:

```
$  for m in gdb insight dejagnu
do
( mkdir -p $m && cd $m && cvs -q -f -d /cvs/src co -P -r $b $m )
done
$
```

Note:

- The reading of '.cvsrc' is disabled ('-f') so that there isn't any confusion between what is written here and what your local cvs really does.

Update relevant files.

'gdb/NEWS'

Major releases get their comments added as part of the mainline. Minor releases should probably mention any significant bugs that were fixed.

Don't forget to include the 'ChangeLog' entry.

```
$  emacs gdb/src/gdb/NEWS
...
c-x 4 a
...
c-x c-s c-x c-c
$  cp gdb/src/gdb/NEWS insight/src/gdb/NEWS
$  cp gdb/src/gdb/ChangeLog insight/src/gdb/ChangeLog
```

'gdb/README'

You'll need to update:

- the version
- the update date
- who did it

```
$  emacs gdb/src/gdb/README
...
c-x 4 a
...
c-x c-s c-x c-c
$  cp gdb/src/gdb/README insight/src/gdb/README
$  cp gdb/src/gdb/ChangeLog insight/src/gdb/ChangeLog
```

Maintainer note: Hopefully the 'README' file was reviewed before the initial branch was cut so just a simple substitute is needed to get it updated.

Maintainer note: Other projects generate 'README' and 'INSTALL' from the core documentation. This might be worth pursuing.

'gdb/version.in'

```
$  echo $v > gdb/src/gdb/version.in
$  cat gdb/src/gdb/version.in
5.2
$  emacs gdb/src/gdb/version.in
...
c-x 4 a
... Bump to version ...
c-x c-s c-x c-c
$  cp gdb/src/gdb/version.in insight/src/gdb/version.in
$  cp gdb/src/gdb/ChangeLog insight/src/gdb/ChangeLog
```

'dejagnu/src/dejagnu/configure.in'

Dejagnu is more complicated. The version number is a parameter to `AM_INIT_AUTOMAKE`. Tweak it to read something like gdb-5.1.91.

Don't forget to re-generate 'configure'.

Don't forget to include a 'ChangeLog' entry.

```
$  emacs dejagnu/src/dejagnu/configure.in
...
c-x 4 a
...
c-x c-s c-x c-c
$  ( cd  dejagnu/src/dejagnu && autoconf )
```

Do the dirty work

This is identical to the process used to create the daily snapshot.

```
$  for m in gdb insight
do
( cd $m/src && gmake -f Makefile.in $m.tar )
done
$  ( m=dejagnu; cd $m/src && gmake -f Makefile.in $m.tar.bz2 )
```

Check the source files

You're looking for files that have mysteriously disappeared. *distclean* has the habit of deleting files it shouldn't. Watch out for the 'version.in' update *cronjob*.

```
$ ( cd gdb/src && cvs -f -q -n update )
M djunpack.bat
? gdb-5.1.91.tar
? proto-toplev
... lots of generated files ...
M gdb/ChangeLog
M gdb/NEWS
M gdb/README
M gdb/version.in
... lots of generated files ...
$
```

Don't worry about the 'gdb.info-??' or 'gdb/p-exp.tab.c'. They were generated (and yes 'gdb.info-1' was also generated only something strange with CVS means that they didn't get supressed). Fixing it would be nice though.

Create compressed versions of the release

```
$ cp */src/*.tar .
$ cp */src/*.bz2 .
$ ls -F
dejagnu/ dejagnu-gdb-5.2.tar.bz2 gdb/ gdb-5.2.tar insight/ insight-5.2.tar
$ for m in gdb insight
do
bzip2 -v -9 -c $m-$v.tar > $m-$v.tar.bz2
gzip -v -9 -c $m-$v.tar > $m-$v.tar.gz
done
$
```

Note:

- A pipe such as *bunzip2 < xxx.bz2 | gzip -9 > xxx.gz* is not since, in that mode, `gzip` does not know the name of the file and, hence, can not include it in the compressed file. This is also why the release process runs `tar` and `bzip2` as separate passes.

15.7.2 Sanity check the tar ball

Pick a popular machine (Solaris/PPC?) and try the build on that.

```
$ bunzip2 < gdb-5.2.tar.bz2 | tar xpf -
$ cd gdb-5.2
$ ./configure
$ make
...
$ ./gdb/gdb ./gdb/gdb
GNU gdb 5.2
...
(gdb)  b main
```

```
Breakpoint 1 at 0x80732bc: file main.c, line 734.
(gdb)  run
Starting program: /tmp/gdb-5.2/gdb/gdb

Breakpoint 1, main (argc=1, argv=0xbffff8b4) at main.c:734
734          catch_errors (captured_main, &args, "", RETURN_MASK_ALL);
(gdb)  print args
$1 = {argc = 136426532, argv = 0x821b7f0}
(gdb)
```

15.7.3 Make a release candidate available

If this is a release candidate then the only remaining steps are:

1. Commit 'version.in' and 'ChangeLog'

2. Tweak 'version.in' (and 'ChangeLog' to read *L.M.N*-0000-00-00-cvs so that the version update process can restart.

3. Make the release candidate available in ftp://sources.redhat.com/pub/gdb/snapshots/branch

4. Notify the relevant mailing lists (gdb@sources.redhat.com and gdb-testers@sources.redhat.com that the candidate is available.

15.7.4 Make a formal release available

(And you thought all that was required was to post an e-mail.)

Install on sware

Copy the new files to both the release and the old release directory:

```
$  cp *.bz2 *.gz ~ftp/pub/gdb/old-releases/
$  cp *.bz2 *.gz ~ftp/pub/gdb/releases
```

Clean up the releases directory so that only the most recent releases are available (e.g. keep 5.2 and 5.2.1 but remove 5.1):

```
$  cd ~ftp/pub/gdb/releases
$  rm ...
```

Update the file 'README' and '.message' in the releases directory:

```
$  vi README
...
$  rm -f .message
$  ln README .message
```

Update the web pages.

'htdocs/download/ANNOUNCEMENT'
> This file, which is posted as the official announcement, includes:
> - General announcement
> - News. If making an *M.N*.1 release, retain the news from earlier *M.N* release.
> - Errata

'`htdocs/index.html`'
'`htdocs/news/index.html`'
'`htdocs/download/index.html`'
  These files include:

- announcement of the most recent release
- news entry (remember to update both the top level and the news directory).

  These pages also need to be regenerate using `index.sh`.

'`download/onlinedocs/`'
  You need to find the magic command that is used to generate the online docs from the '`.tar.bz2`'. The best way is to look in the output from one of the nightly `cron` jobs and then just edit accordingly. Something like:

```
$  ~/ss/update-web-docs \
 ~ftp/pub/gdb/releases/gdb-5.2.tar.bz2 \
 $PWD/www \
 /www/sourceware/htdocs/gdb/download/onlinedocs \
 gdb
```

'`download/ari/`'
  Just like the online documentation. Something like:

```
$  /bin/sh ~/ss/update-web-ari \
 ~ftp/pub/gdb/releases/gdb-5.2.tar.bz2 \
 $PWD/www \
 /www/sourceware/htdocs/gdb/download/ari \
 gdb
```

Shadow the pages onto gnu

Something goes here.

Install the GDB tar ball on GNU

At the time of writing, the GNU machine was *gnudist.gnu.org* in '`~ftp/gnu/gdb`'.

Make the '`ANNOUNCEMENT`'

Post the '`ANNOUNCEMENT`' file you created above to:

- GDB Announcement mailing list
- General GNU Announcement list (but delay it a day or so to let things get out)
- GDB Bug Report mailing list

15.7.5 Cleanup

The release is out but you're still not finished.

Commit outstanding changes

In particular you'll need to commit any changes to:

- 'gdb/ChangeLog'
- 'gdb/version.in'
- 'gdb/NEWS'
- 'gdb/README'

Tag the release

Something like:
```
$  d=`date -u +%Y-%m-%d`
$  echo $d
2002-01-24
$  ( cd insight/src/gdb && cvs -f -q update )
$  ( cd insight/src && cvs -f -q tag gdb_5_2-$d-release )
```
Insight is used since that contains more of the release than GDB (`dejagnu` doesn't get tagged but I think we can live with that).

Mention the release on the trunk

Just put something in the 'ChangeLog' so that the trunk also indicates when the release was made.

Restart 'gdb/version.in'

If 'gdb/version.in' does not contain an ISO date such as *2002-01-24* then the daily `cronjob` won't update it. Having committed all the release changes it can be set to '5.2.0_0000-00-00-cvs' which will restart things (yes the _ is important - it affects the snapshot process).

Don't forget the 'ChangeLog'.

Merge into trunk

The files committed to the branch may also need changes merged into the trunk.

Revise the release schedule

Post a revised release schedule to GDB Discussion List with an updated announcement. The schedule can be generated by running:
```
$  ~/ss/schedule `date +%s` schedule
```
The first parameter is approximate date/time in seconds (from the epoch) of the most recent release.

Also update the schedule `cronjob`.

15.8 Post release

Remove any `OBSOLETE` code.

16 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.

16.1 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.

16.2 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 any language lang 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.

16.3 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.

17 Hints

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

17.1 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 have 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.

17.2 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.

17.3 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 bureaucracy 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., 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.

17.4 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!

Appendix A GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that

could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

   D. Preserve all the copyright notices of the Document.

   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

    You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

    The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

    In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

    You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

    You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

    A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

    If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year   your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled "GNU
Free Documentation License."
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being list"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

H

I

K

L

M

N

O