GDB Internals

A guide to the internals of the GNU debugger

John Gilmore
Cygnus Solutions
Second Edition:
Stan Shebs
Cygnus Solutions

Cygnus Solutions
Revision
TEXinfo 2003-02-03.16

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 Summary

1.1 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.

1.2 Contributors

The first edition of this document was written by John Gilmore of Cygnus Solutions. The current second edition was written by Stan Shebs of Cygnus Solutions, who continues to update the manual.

Over the years, many others have made additions and changes to this document. This section attempts to record the significant contributors to that effort. One of the virtues of free software is that everyone is free to contribute to it; with regret, we cannot actually acknowledge everyone here.

> *Plea:* This section has only been added relatively recently (four years after publication of the second edition). Additions to this section are particularly welcome. If you or your friends (or enemies, to be evenhanded) have been unfairly omitted from this list, we would like to add your names!

A document such as this relies on being kept up to date by numerous small updates by contributing engineers as they make changes to the code base. The file '`ChangeLog`' in the GDB distribution approximates a blow-by-blow account. The most prolific contributors to this important, but low profile task are Andrew Cagney (responsible for over half the entries), Daniel Jacobowitz, Mark Kettenis, Jim Blandy and Eli Zaretskii.

Eli Zaretskii and Daniel Jacobowitz wrote the sections documenting watchpoints.

Jeremy Bennett updated the sections on initializing a new architecture and register representation, and added the section on Frame Interpretation.

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. These are all calculated by `autoconf` when the debugger is built.

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, calling `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 as the traditional native debugger on a system, you will need to supply both target and native information.

2.4 Source Tree Structure

The GDB source directory has a mostly flat structure—there are only a few subdirectories. A file's name usually gives a hint as to what it does; for example, '`stabsread.c`' reads stabs, '`dwarf2read.c`' reads DWARF 2, etc.

Files that are related to some common task have names that share common substrings. For example, '`*-thread.c`' files deal with debugging threads on various platforms; '`*read.c`' files deal with reading various kinds of symbol and object files; '`inf*.c`' files deal with direct control of the *inferior program* (GDB parlance for the program being debugged).

There are several dozens of files in the '`*-tdep.c`' family. '`tdep`' stands for *target-dependent code*—each of these files implements debug support for a specific target architecture (sparc, mips, etc). Usually, only one of these will be used in a specific GDB configuration (sometimes two, closely related).

Similarly, there are many '`*-nat.c`' files, each one for native debugging on a specific system (e.g., '`sparc-linux-nat.c`' is for native debugging of Sparc machines running the Linux kernel).

The few subdirectories of the source tree are:

'`cli`' Code that implements *CLI*, the GDB Command-Line Interpreter. See Chapter 4 [User Interface], page 14.

'`gdbserver`'
Code for the GDB remote server.

'`gdbtk`' Code for Insight, the GDB TK-based GUI front-end.

‘`mi`’	The *GDB/MI*, the GDB Machine Interface interpreter.
‘`signals`’	Target signal translation code.
‘`tui`’	Code for *TUI*, the GDB Text-mode full-screen User Interface. See Chapter 4 [User Interface], page 14.

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 Prologue Analysis

To produce a backtrace and allow the user to manipulate older frames' variables and arguments, GDB needs to find the base addresses of older frames, and discover where those frames' registers have been saved. Since a frame's "callee-saves" registers get saved by younger frames if and when they're reused, a frame's registers may be scattered unpredictably across younger frames. This means that changing the value of a register-allocated variable in an older frame may actually entail writing to a save slot in some younger frame.

Modern versions of GCC emit Dwarf call frame information ("CFI"), which describes how to find frame base addresses and saved registers. But CFI is not always available, so as a fallback GDB uses a technique called *prologue analysis* to find frame sizes and saved registers. A prologue analyzer disassembles the function's machine code starting from its entry point, and looks for instructions that allocate frame space, save the stack pointer in a frame pointer register, save registers, and so on. Obviously, this can't be done accurately in general, but it's tractable to do well enough to be very helpful. Prologue analysis predates the GNU toolchain's support for CFI; at one time, prologue analysis was the only mechanism GDB used for stack unwinding at all, when the function calling conventions didn't specify a fixed frame layout.

In the olden days, function prologues were generated by hand-written, target-specific code in GCC, and treated as opaque and untouchable by optimizers. Looking at this code, it was usually straightforward to write a prologue analyzer for GDB that would accurately understand all the prologues GCC would generate. However, over time GCC became more aggressive about instruction scheduling, and began to understand more about the semantics of the prologue instructions themselves; in response, GDB's analyzers became more complex and fragile. Keeping the prologue analyzers working as GCC (and the instruction sets themselves) evolved became a substantial task.

To try to address this problem, the code in ‘`prologue-value.h`’ and ‘`prologue-value.c`’ provides a general framework for writing prologue analyzers that are simpler and more robust than ad-hoc analyzers. When we analyze a prologue using the prologue-value framework, we're really doing "abstract interpretation" or "pseudo-evaluation": running the function's code in simulation, but using conservative approximations of the values registers and memory would hold when the code actually runs. For example, if our function starts with the instruction:

```
addi r1, 42     # add 42 to r1
```

we don't know exactly what value will be in `r1` after executing this instruction, but we do know it'll be 42 greater than its original value.

If we then see an instruction like:

```
addi r1, 22     # add 22 to r1
```

we still don't know what `r1's` value is, but again, we can say it is now 64 greater than its original value.

If the next instruction were:

```
mov r2, r1      # set r2 to r1's value
```

then we can say that `r2's` value is now the original value of `r1` plus 64.

It's common for prologues to save registers on the stack, so we'll need to track the values of stack frame slots, as well as the registers. So after an instruction like this:

```
mov (fp+4), r2
```

then we'd know that the stack slot four bytes above the frame pointer holds the original value of `r1` plus 64.

And so on.

Of course, this can only go so far before it gets unreasonable. If we wanted to be able to say anything about the value of `r1` after the instruction:

```
xor r1, r3      # exclusive-or r1 and r3, place result in r1
```

then things would get pretty complex. But remember, we're just doing a conservative approximation; if exclusive-or instructions aren't relevant to prologues, we can just say `r1`'s value is now "unknown". We can ignore things that are too complex, if that loss of information is acceptable for our application.

So when we say "conservative approximation" here, what we mean is an approximation that is either accurate, or marked "unknown", but never inaccurate.

Using this framework, a prologue analyzer is simply an interpreter for machine code, but one that uses conservative approximations for the contents of registers and memory instead of actual values. Starting from the function's entry point, you simulate instructions up to the current PC, or an instruction that you don't know how to simulate. Now you can examine the state of the registers and stack slots you've kept track of.

- To see how large your stack frame is, just check the value of the stack pointer register; if it's the original value of the SP minus a constant, then that constant is the stack frame's size. If the SP's value has been marked as "unknown", then that means the prologue has done something too complex for us to track, and we don't know the frame size.
- To see where we've saved the previous frame's registers, we just search the values we've tracked — stack slots, usually, but registers, too, if you want — for something equal to the register's original value. If the calling conventions suggest a standard place to save a given register, then we can check there first, but really, anything that will get us back the original value will probably work.

This does take some work. But prologue analyzers aren't quick-and-simple pattern patching to recognize a few fixed prologue forms any more; they're big, hairy functions.

Along with inferior function calls, prologue analysis accounts for a substantial portion of the time needed to stabilize a GDB port. So it's worthwhile to look for an approach that will be easier to understand and maintain. In the approach described above:

- It's easier to see that the analyzer is correct: you just see whether the analyzer properly (albeit conservatively) simulates the effect of each instruction.
- It's easier to extend the analyzer: you can add support for new instructions, and know that you haven't broken anything that wasn't already broken before.
- It's orthogonal: to gather new information, you don't need to complicate the code for each instruction. As long as your domain of conservative values is already detailed enough to tell you what you need, then all the existing instruction simulations are already gathering the right data for you.

The file '`prologue-value.h`' contains detailed comments explaining the framework and how to use it.

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 anything 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.

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

`target_remove_breakpoint (`*`bp_tgt`*`)`
`target_insert_breakpoint (`*`bp_tgt`*`)`

Insert or remove a software breakpoint at address *`bp_tgt`*`->placed_address`. Returns zero for success, non-zero for failure. On input, *bp_tgt* contains the address of the breakpoint, and is otherwise initialized to zero. The fields of the `struct bp_target_info` pointed to by *bp_tgt* are updated to contain other information about the breakpoint on output. The field `placed_address` may be updated if the breakpoint was placed at a related address; the field `shadow_contents` contains the real contents of the bytes where the breakpoint has been inserted, if reading memory would return the breakpoint instead of the underlying memory; the field `shadow_len` is the length of memory cached in `shadow_contents`, if any; and the field `placed_size` is optionally set and used by the target, if it could differ from `shadow_len`.

For example, the remote target '`Z0`' packet does not require shadowing memory, so `shadow_len` is left at zero. However, the length reported by `gdbarch_breakpoint_from_pc` is cached in `placed_size`, so that a matching '`z0`' packet can be used to remove the breakpoint.

`target_remove_hw_breakpoint (`*`bp_tgt`*`)`
`target_insert_hw_breakpoint (`*`bp_tgt`*`)`

Insert or remove a hardware-assisted breakpoint at address *`bp_tgt`*`->placed_address`. Returns zero for success, non-zero for failure. See `target_insert_breakpoint` for a description of the `struct bp_target_info` pointed to by *bp_tgt*; the `shadow_contents` and `shadow_len` members are not used for hardware breakpoints, but `placed_size` may be.

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 function called `gdbarch_get_longjmp_target`, which will examine the `jmp_buf` structure and extract the `longjmp` target address. Since `jmp_buf` is target specific and typically defined in a target header not available to GDB, you will need to determine the offset of the PC manually and return that; many targets define a `jb_pc_offset` field in the tdep structure to save the value once calculated.

3.8 Watchpoints

Watchpoints are a special kind of breakpoints (see Chapter 3 [Algorithms], page 4) 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.

When the inferior stops, GDB tries to establish, among other possible reasons, whether it stopped due to a watchpoint being hit. It first uses `STOPPED_BY_WATCHPOINT` to see if any watchpoint was hit. If not, all watchpoint checking is skipped.

Then GDB calls `target_stopped_data_address` exactly once. This method returns the address of the watchpoint which triggered, if the target can determine it. If the triggered address is available, GDB compares the address returned by this method with each watched memory address in each active watchpoint. For data-read and data-access watchpoints, GDB announces every watchpoint that watches the triggered address as being hit. For this reason, data-read and data-access watchpoints *require* that the triggered address be available; if not, read and access watchpoints will never be considered hit. For data-write watchpoints, if the triggered address is available, GDB considers only those watchpoints which match that address; otherwise, GDB considers all data-write watchpoints. For each data-write watchpoint that GDB considers, it evaluates the expression whose value is being watched, and tests whether the watched value has changed. Watchpoints whose watched values have changed are announced as hit.

GDB uses several macros and primitives to support 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_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_stopped_data_address (`*`addr_p`*`)`
: If the inferior has some watchpoint that triggered, place the address associated with the watchpoint at the location pointed to by *addr_p* and return non-zero. Otherwise, return zero. This is required for data-read and data-access watchpoints. It is not required for data-write watchpoints, but GDB uses it to improve handling of those also.

  GDB will only call this method once per watchpoint stop, immediately after calling `STOPPED_BY_WATCHPOINT`. If the target's watchpoint indication is sticky, i.e., stays set after resuming, this method should clear it. For instance, the x86 debug control register has sticky triggered flags.

`target_watchpoint_addr_within_range (target, addr, start, length)`
: Check whether *addr* (as returned by `target_stopped_data_address`) lies within the hardware-defined watchpoint region described by *start* and *length*. This only needs to be provided if the granularity of a watchpoint is greater than one byte, i.e., if the watchpoint can also trigger on nearby addresses outside of the watched region.

`HAVE_STEPPABLE_WATCHPOINT`
: If defined to a non-zero value, it is not necessary to disable a watchpoint to step over it. Like `gdbarch_have_nonsteppable_watchpoint`, this is usually set when watchpoints trigger at the instruction which will perform an interesting read or write. It should be set if there is a temporary disable bit which allows the processor to step over the interesting instruction without raising the watchpoint exception again.

`int gdbarch_have_nonsteppable_watchpoint (gdbarch)`
: If it returns a non-zero value, GDB should disable a watchpoint to step the inferior over it. This is usually set when watchpoints trigger at the instruction which will perform an interesting read or write.

`HAVE_CONTINUABLE_WATCHPOINT`
: If defined to a non-zero value, it is possible to continue the inferior after a watchpoint has been hit. This is usually set when watchpoints trigger at the instruction following an interesting read or write.

`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`'. Normally, this macro is defined to invoke the function pointed to by the `to_stopped_by_watchpoint` member of the structure (of the type `target_ops`, defined on '`target.h`') that describes the target-specific operations; `to_stopped_by_watchpoint` ignores the *wait_status* argument.

  GDB does not require the non-zero value returned by `STOPPED_BY_WATCHPOINT` to be 100% correct, so if a target cannot determine for sure whether the inferior stopped due to a watchpoint, it could return non-zero "just in case".

3.8.1 Watchpoints and Threads

GDB only supports process-wide watchpoints, which trigger in all threads. GDB uses the thread ID to make watchpoints act as if they were thread-specific, but it cannot set hardware watchpoints that only trigger in a specific thread. Therefore, even if the target supports threads, per-thread debug registers, and watchpoints which only affect a single thread, it should set the per-thread debug registers for all threads to the same value. On GNU/Linux native targets, this is accomplished by using `ALL_LWPS` in `target_insert_watchpoint` and `target_remove_watchpoint` and by using `linux_set_new_thread` to register a handler for newly created threads.

GDB's GNU/Linux support only reports a single event at a time, although multiple events can trigger simultaneously for multi-threaded programs. When multiple events occur, '`linux-nat.c`' queues subsequent events and returns them the next time the program is resumed. This means that `STOPPED_BY_WATCHPOINT` and `target_stopped_data_address` only need to consult the current thread's state—the thread indicated by `inferior_ptid`. If two threads have hit watchpoints simultaneously, those routines will be called a second time for the second thread.

3.8.2 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.

(At present, the library functions read and write debug registers directly, and are thus only available for native configurations.)

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 14 [Native Debugging], page 73).
- 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 (addr_p)`
: The target function `target_stopped_data_address` is set to call this function. 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_stopped_by_watchpoint (void)`
: The macro `STOPPED_BY_WATCHPOINT` is 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 true if any bit is set. Otherwise, false is returned.

`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 (`*`bp_tgt`*`)`
`i386_remove_hw_breakpoint (`*`bp_tgt`*`)`
: 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. The argument is a `struct bp_target_info *`, as described in the documentation for `target_insert_breakpoint`. 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_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.

3.9 Checkpoints

In the abstract, a checkpoint is a point in the execution history of the program, which the user may wish to return to at some later time.

Internally, a checkpoint is a saved copy of the program state, including whatever information is required in order to restore the program to that state at a later time. This can be expected to include the state of registers and memory, and may include external state such as the state of open files and devices.

There are a number of ways in which checkpoints may be implemented in gdb, e.g. as corefiles, as forked processes, and as some opaque method implemented on the target side.

A corefile can be used to save an image of target memory and register state, which can in principle be restored later — but corefiles do not typically include information about external entities such as open files. Currently this method is not implemented in gdb.

A forked process can save the state of user memory and registers, as well as some subset of external (kernel) state. This method is used to implement checkpoints on Linux, and in principle might be used on other systems.

Some targets, e.g. simulators, might have their own built-in method for saving checkpoints, and gdb might be able to take advantage of that capability without necessarily knowing any details of how it is done.

3.10 Observing changes in GDB internals

In order to function properly, several modules need to be notified when some changes occur in the GDB internals. Traditionally, these modules have relied on several paradigms, the most common ones being hooks and gdb-events. Unfortunately, none of these paradigms was versatile enough to become the standard notification mechanism in GDB. The fact that they only supported one "client" was also a strong limitation.

A new paradigm, based on the Observer pattern of the *Design Patterns* book, has therefore been implemented. The goal was to provide a new interface overcoming the issues with the notification mechanisms previously available. This new interface needed to be strongly typed, easy to extend, and versatile enough to be used as the standard interface when adding new notifications.

See Appendix A [GDB Observers], page 109 for a brief description of the observers currently implemented in GDB. The rationale for the current implementation is also briefly discussed.

4 User Interface

GDB has several user interfaces, of which the traditional command-line interface is perhaps the most familiar.

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` *nr_rows*`, const char *`*tblid*) [Function]

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 ui_align` *alignment*`, const char *`*colhdr*) [Function]

`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 ui_out *uiout, const char *id`) [Function]

This function first opens the tuple and then establishes a cleanup (see Chapter 16 [Coding], page 80) 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);
```

`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 ui_out *uiout, const char *id`) [Function]

Similar to `make_cleanup_ui_out_tuple_begin_end`, this function opens a list and then establishes cleanup (see Chapter 16 [Coding], page 80) that will close the 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 *format, ...`) [Function]

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, int value`) [Function]

This function outputs a value of an `int` variable. It uses the `"%d"` output conversion specification. *fldname* specifies the name of the field.

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

`void` **ui_out_field_fmt_int** (`struct ui_out *uiout, int width, enum ui_align alignment, const char *fldname, int value`) [Function]

This function outputs a value of an `int` variable. It differs from `ui_out_field_int` in that the caller specifies the desired *width* and *alignment* of the output. *fldname* specifies the name of the field.

`void` **ui_out_field_core_addr** (`struct ui_out *uiout, const char *fldname, struct gdbarch *gdbarch, CORE_ADDR address`) [Function]

This function outputs an address as appropriate for *gdbarch*.

`void` **ui_out_field_string** (`struct ui_out *uiout, const char *fldname, const char *string`) [Function]

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 *fieldname, struct ui_stream *streambuf`) [Function]

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 char *`*format*`, ...`*`) [Function]

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.

[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.

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 ();

    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 (print_address_bits <= 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 ",
      hex_string_custom ((unsigned long) b->address, 8));
```

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 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 Values

6.1 Values

GDB uses struct value, or *values*, as an internal abstraction for the representation of a variety of inferior objects and GDB convenience objects.

Values have an associated struct type, that describes a virtual view of the raw data or object stored in or accessed through the value.

A value is in addition discriminated by its lvalue-ness, given its enum lval_type enumeration type:

not_lval This value is not an lval. It can't be assigned to.

lval_memory
: This value represents an object in memory.

lval_register
: This value represents an object that lives in a register.

lval_internalvar
: Represents the value of an internal variable.

lval_internalvar_component
: Represents part of a GDB internal variable. E.g., a structure field.

lval_computed
: These are "computed" values. They allow creating specialized value objects for specific purposes, all abstracted away from the core value support code. The creator of such a value writes specialized functions to handle the reading and writing to/from the value's backend data, and optionally, a "copy operator" and a "destructor".

  Pointers to these functions are stored in a struct lval_funcs instance (declared in 'value.h'), and passed to the allocate_computed_value function, as in the example below.

```
static void
nil_value_read (struct value *v)
{
  /* This callback reads data from some backend, and stores it in V.
     In this case, we always read null data.  You'll want to fill in
     something more interesting.  */

  memset (value_contents_all_raw (v),
          value_offset (v),
```

```
          TYPE_LENGTH (value_type (v)));
}

static void
nil_value_write (struct value *v, struct value *fromval)
{
  /* Takes the data from FROMVAL and stores it in the backend of V.  */

  to_oblivion (value_contents_all_raw (fromval),
               value_offset (v),
               TYPE_LENGTH (value_type (fromval)));
}

static struct lval_funcs nil_value_funcs =
  {
    nil_value_read,
    nil_value_write
  };

struct value *
make_nil_value (void)
{
   struct type *type;
   struct value *v;

   type = make_nils_type ();
   v = allocate_computed_value (type, &nil_value_funcs, NULL);

   return v;
}
```

See the implementation of the `$_siginfo` convenience variable in '`infrun.c`' as a real example use of lval_computed.

7 Stack Frames

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

GDB's frame model, a fresh design, was implemented with the need to support DWARF's Call Frame Information in mind. In fact, the term "unwind" is taken directly from that specification. Developers wishing to learn more about unwinders, are encouraged to read the DWARF specification, available from `http://www.dwarfstd.org`.

GDB's model is that you find a frame's registers by "unwinding" them from the next younger frame. That is, '`get_frame_register`' which returns the value of a register in frame #1 (the next-to-youngest frame), is implemented by calling frame #0's `frame_register_unwind` (the youngest frame). But then the obvious question is: how do you access the registers of the youngest frame itself?

To answer this question, GDB has the *sentinel* frame, the "-1st" frame. Unwinding registers from the sentinel frame gives you the current values of the youngest real frame's registers. If f is a sentinel frame, then `get_frame_type (f)` $\equiv$ `SENTINEL_FRAME`.

7.1 Selecting an Unwinder

The architecture registers a list of frame unwinders (`struct frame_unwind`), using the functions `frame_unwind_prepend_unwinder` and `frame_unwind_append_unwinder`. Each unwinder includes a sniffer. Whenever GDB needs to unwind a frame (to fetch the previous frame's registers or the current frame's ID), it calls registered sniffers in order to find one which recognizes the frame. The first time a sniffer returns non-zero, the corresponding unwinder is assigned to the frame.

7.2 Unwinding the Frame ID

Every frame has an associated ID, of type `struct frame_id`. The ID includes the stack base and function start address for the frame. The ID persists through the entire life of the frame, including while other called frames are running; it is used to locate an appropriate `struct frame_info` from the cache.

Every time the inferior stops, and at various other times, the frame cache is flushed. Because of this, parts of GDB which need to keep track of individual frames cannot use pointers to `struct frame_info`. A frame ID provides a stable reference to a frame, even when the unwinder must be run again to generate a new `struct frame_info` for the same frame.

The frame's unwinder's `this_id` method is called to find the ID. Note that this is different from register unwinding, where the next frame's `prev_register` is called to unwind this frame's registers.

Both stack base and function address are required to identify the frame, because a recursive function has the same function address for two consecutive frames and a leaf function may have the same stack address as its caller. On some platforms, a third address is part of the ID to further disambiguate frames—for instance, on IA-64 the separate register stack address is included in the ID.

An invalid frame ID (`outer_frame_id`) returned from the `this_id` method means to stop unwinding after this frame.

`null_frame_id` is another invalid frame ID which should be used when there is no frame. For instance, certain breakpoints are attached to a specific frame, and that frame is identified through its frame ID (we use this to implement the "finish" command). Using `null_frame_id` as the frame ID for a given breakpoint means that the breakpoint is not specific to any frame. The `this_id` method should never return `null_frame_id`.

7.3 Unwinding Registers

Each unwinder includes a `prev_register` method. This method takes a frame, an associated cache pointer, and a register number. It returns a `struct value *` describing the requested register, as saved by this frame. This is the value of the register that is current in this frame's caller.

The returned value must have the same type as the register. It may have any lvalue type. In most circumstances one of these routines will generate the appropriate value:

`frame_unwind_got_optimized`
: This register was not saved.

`frame_unwind_got_register`
: This register was copied into another register in this frame. This is also used for unchanged registers; they are "copied" into the same register.

`frame_unwind_got_memory`
: This register was saved in memory.

`frame_unwind_got_constant`
: This register was not saved, but the unwinder can compute the previous value some other way.

`frame_unwind_got_address`
: Same as `frame_unwind_got_constant`, except that the value is a target address. This is frequently used for the stack pointer, which is not explicitly saved but has a known offset from this frame's stack pointer. For architectures with a flat unified address space, this is generally the same as `frame_unwind_got_constant`.

8 Symbol Handling

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

Symbol information for a large program can be truly massive, and reading of symbol information is one of the major performance bottlenecks in GDB; it can take many minutes to process it all. Studies have shown that nearly all the time spent is computational, rather than file reading.

One of the ways for GDB to provide a good user experience is to start up quickly, taking no more than a few seconds. It is simply not possible to process all of a program's debugging info in that time, and so we attempt to handle symbols incrementally. For instance, we create *partial symbol tables* consisting of only selected symbols, and only expand them to full symbol tables when necessary.

8.1 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. In addition, it may bring in more symbols while loading shared libraries.

Symbol files are initially opened by code in '`symfile.c`' using the BFD library (see Chapter 15 [Support Libraries], page 75). 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.

8.2 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.

8.3 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.

8.4 Object File Formats

8.4.1 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`'.

8.4.2 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. GNU's COFF-using configs often use stabs-type info, encapsulated in special sections.

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

8.4.3 ECOFF

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

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

8.4.4 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).

8.4.5 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.

8.4.6 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. Debugging info may be either stabs encapsulated in ELF sections, or more commonly these days, DWARF.

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

8.4.7 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 '`somread.c`'.

8.5 Debugging File Formats

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

8.5.1 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.

8.5.2 COFF

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

8.5.3 Mips debug (Third Eye)

ECOFF includes a definition of a special debug format.

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

8.5.4 DWARF 2

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

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

8.5.5 Compressed DWARF 2

Compressed DWARF 2 is not technically a separate debugging format, but merely DWARF 2 debug information that has been compressed. In this format, every object-file section holding DWARF 2 debugging information is compressed and prepended with a header. (The section is also typically renamed, so a section called `.debug_info` in a DWARF 2 binary would be called `.zdebug_info` in a compressed DWARF 2 binary.) The header is 12 bytes long:

- 4 bytes: the literal string "ZLIB"
- 8 bytes: the uncompressed size of the section, in big-endian byte order.

The same reader is used for both compressed an normal DWARF 2 info. Section decompression is done in `zlib_decompress_section` in '`dwarf2read.c`'.

8.5.6 DWARF 3

DWARF 3 is an improved version of DWARF 2.

8.5.7 SOM

Like COFF, the SOM definition includes debugging information.

8.6 Adding a New Symbol Reader to GDB

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

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

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

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

8.7 Memory Management for Symbol Files

Most memory associated with a loaded symbol file is stored on its `objfile_obstack`. This includes symbols, types, namespace data, and other information produced by the symbol readers.

Because this data lives on the objfile's obstack, it is automatically released when the objfile is unloaded or reloaded. Therefore one objfile must not reference symbol or type data from another objfile; they could be unloaded at different times.

User convenience variables, et cetera, have associated types. Normally these types live in the associated objfile. However, when the objfile is unloaded, those types are deep copied to global memory, so that the values of the user variables and history items are not lost.

9 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.

9.1 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`').

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!

10 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.

10.1 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 is a Makefile fragment that once contained both host and native configuration information (see Chapter 14 [Native Debugging], page 73) for the machine *xyz*. The host configuration information is now handled by Autoconf.

Host configuration information included 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.

(Files named '`gdb/config/`*`arch`*`/xm-`*`xyz`*`.h`' were once used to define host-specific macros, but were no longer needed and have all been removed.)

Generic Host Support Files

There are some "generic" versions of routines that can be used by various systems.

'`ser-unix.c`'
: This contains serial line support for Unix systems. It is included by default on all Unix-like hosts.

'`ser-pipe.c`'
: This contains serial pipe support for Unix systems. It is included by default on all Unix-like hosts.

'`ser-mingw.c`'
: This contains serial line support for 32-bit programs running under Windows using MinGW.

'`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. It is included by default on all Unix-like hosts and with MinGW.

10.2 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. While formerly they could be set in host-specific header files, at present they can be changed only by setting `CFLAGS` when building, or by editing the source code.

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:

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

`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`.

`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"`.

`ISATTY` Substitute for isatty, if not available.

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

`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.

`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.

`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.

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

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

11 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`'.

11.1 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 '`defs.h`':

`GDB_OSABI_UNINITIALIZED`
: Used for struct gdbarch_info if ABI is still uninitialized.

`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_OPENBSD_ELF`
: OpenBSD using the ELF executable format.

`GDB_OSABI_WINCE`
: Windows CE.

`GDB_OSABI_GO32`
: DJGPP.

`GDB_OSABI_IRIX`
: Irix.

`GDB_OSABI_INTERIX`
: Interix (Posix layer for MS-Windows systems).

`GDB_OSABI_HPUX_ELF`
: HP/UX using the ELF executable format.

`GDB_OSABI_HPUX_SOM`
: HP/UX using the SOM executable format.

`GDB_OSABI_QNXNTO`
: QNX Neutrino.

`GDB_OSABI_CYGWIN`
: Cygwin.

`GDB_OSABI_AIX`
: AIX.

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*, `unsigned long` *machine*, `enum gdb_osabi` *osabi*, `void` (`*`*init_osabi*)(`struct gdbarch_info` *info*, `struct gdbarch *`*gdbarch*)) [Function]
Register the OS ABI handler specified by *init_osabi* for the architecture, machine type and OS ABI specified by *arch*, *machine* and *osabi*. In most cases, a value of zero for the machine type, which implies the architecture's default machine type, will suffice.

`void` **gdbarch_register_osabi_sniffer** (`enum bfd_architecture` *arch*, `enum bfd_flavour` *flavour*, `enum gdb_osabi` (`*`*sniffer*)(`bfd *`*abfd*)) [Function]
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 considered to be generic, and is allowed to examine *flavour*-flavoured files for any architecture.

`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 gdbarch *`*gdbarch*, `enum gdb_osabi` *osabi*) [Function]
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*.

`void` **generic_elf_osabi_sniff_abi_tag_sections** (`bfd *`*abfd*, `asection *`*sect*, `void *`*obj*) [Function]
Helper routine for ELF file sniffers. Examine the file described by *abfd* and look at ABI tag note sections to determine the OS ABI from the note. This function should be called via `bfd_map_over_sections`.

11.2 Initializing a New Architecture

11.2.1 How an Architecture is Represented

Each `gdbarch` is associated with a single BFD architecture, via a `bfd_arch_`*`arch`* in the `bfd_architecture` enumeration. The `gdbarch` is registered by a call to `register_gdbarch_init`, usually from the file's `_initialize_`*`filename`* routine, which will be automatically called during GDB startup. The arguments are a BFD architecture constant and an initialization function.

A GDB description for a new architecture, *arch* is created by defining a global function `_initialize_`*`arch`*`_tdep`, by convention in the source file '*`arch`*`-tdep.c`'. For example, in the case of the OpenRISC 1000, this function is called `_initialize_or1k_tdep` and is found in the file '`or1k-tdep.c`'.

The resulting object files containing the implementation of the `_initialize_`*`arch`*`_tdep` function are specified in the GDB '`configure.tgt`' file, which includes a large case statement pattern matching against the `--target` option of the `configure` script. The new `struct gdbarch` is created within the `_initialize_`*`arch`*`_tdep` function by calling `gdbarch_register`:

```
void gdbarch_register (enum bfd_architecture    architecture,
                       gdbarch_init_ftype      *init_func,
                       gdbarch_dump_tdep_ftype *tdep_dump_func);
```

The *architecture* will identify the unique BFD to be associated with this `gdbarch`. The *init_func* funciton is called to create and return the new `struct gdbarch`. The *tdep_dump_func* function will dump the target specific details associated with this architecture.

For example the function `_initialize_or1k_tdep` creates its architecture for 32-bit OpenRISC 1000 architectures by calling:

```
gdbarch_register (bfd_arch_or32, or1k_gdbarch_init, or1k_dump_tdep);
```

11.2.2 Looking Up an Existing Architecture

The initialization function has this prototype:

```
static struct gdbarch *
arch_gdbarch_init (struct gdbarch_info info,
                   struct gdbarch_list *arches)
```

The *info* argument contains parameters used to select the correct architecture, and *arches* is a list of architectures which have already been created with the same `bfd_arch_`*`arch`* value.

The initialization function should first make sure that *info* is acceptable, and return `NULL` if it is not. Then, it should search through *arches* for an exact match to *info*, and return one if found. Lastly, if no exact match was found, it should create a new architecture based on *info* and return it.

The lookup is done using `gdbarch_list_lookup_by_info`. It is passed the list of existing architectures, *arches*, and the `struct gdbarch_info`, *info*, and returns the first matching architecture it finds, or `NULL` if none are found. If an architecture is found it can be returned

as the result from the initialization function, otherwise a new `struct gdbach` will need to be created.

The struct gdbarch_info has the following components:

```
struct gdbarch_info
{
   const struct bfd_arch_info *bfd_arch_info;
   int                         byte_order;
   bfd                        *abfd;
   struct gdbarch_tdep_info   *tdep_info;
   enum gdb_osabi              osabi;
   const struct target_desc   *target_desc;
};
```

The `bfd_arch_info` member holds the key details about the architecture. The `byte_order` member is a value in an enumeration indicating the endianism. The `abfd` member is a pointer to the full BFD, the `tdep_info` member is additional custom target specific information, `osabi` identifies which (if any) of a number of operating specific ABIs are used by this architecture and the `target_desc` member is a set of name-value pairs with information about register usage in this target.

When the `struct gdbarch` initialization function is called, not all the fields are provided—only those which can be deduced from the BFD. The `struct gdbarch_info`, *info* is used as a look-up key with the list of existing architectures, *arches* to see if a suitable architecture already exists. The *tdep_info*, *osabi* and *target_desc* fields may be added before this lookup to refine the search.

Only information in *info* should be used to choose the new architecture. Historically, *info* could be sparse, and defaults would be collected from the first element on *arches*. However, GDB now fills in *info* more thoroughly, so new `gdbarch` initialization functions should not take defaults from *arches*.

11.2.3 Creating a New Architecture

If no architecture is found, then a new architecture must be created, by calling `gdbarch_alloc` using the supplied `struct gdbarch_info` and any additional custom target specific information in a `struct gdbarch_tdep`. The prototype for `gdbarch_alloc` is:

```
struct gdbarch *gdbarch_alloc (const struct gdbarch_info *info,
                               struct gdbarch_tdep       *tdep);
```

The newly created struct gdbarch must then be populated. Although there are default values, in most cases they are not what is required.

For each element, *X*, there is are a pair of corresponding accessor functions, one to set the value of that element, `set_gdbarch_X`, the second to either get the value of an element (if it is a variable) or to apply the element (if it is a function), `gdbarch_X`. Note that both accessor functions take a pointer to the `struct gdbarch` as first argument. Populating the new `gdbarch` should use the `set_gdbarch` functions.

The following sections identify the main elements that should be set in this way. This is not the complete list, but represents the functions and elements that must commonly be specified for a new architecture. Many of the functions and variables are described in the header file '`gdbarch.h`'.

This is the main work in defining a new architecture. Implementing the set of functions to populate the `struct gdbarch`.

`struct gdbarch_tdep` is not defined within GDB—it is up to the user to define this struct if it is needed to hold custom target information that is not covered by the standard `struct gdbarch`. For example with the OpenRISC 1000 architecture it is used to hold the number of matchpoints available in the target (along with other information).

If there is no additional target specific information, it can be set to `NULL`.

11.3 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 `gdbarch_register_name` and related functions.

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

11.4 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 Renesas 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.)

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

[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.

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*`, CORE_ADDR` *addr*) [Function]

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` *addr*) [Function]

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`.

Here are two functions 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` **gdbarch_pointer_to_address** (`struct gdbarch *gdbarch, struct type *type, char *buf`) [Function]

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` **gdbarch_address_to_pointer** (`struct gdbarch *gdbarch, struct type *type, char *buf, CORE_ADDR addr`) [Function]

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.

11.5 Address Classes

Sometimes information about different kinds of addresses is available via the debug information. For example, some programming environments define addresses of several different sizes. If the debug information distinguishes these kinds of address classes through either the size info (e.g, `DW_AT_byte_size` in DWARF 2) or through an explicit address class attribute (e.g, `DW_AT_address_class` in DWARF 2), the following macros should be defined in order to disambiguate these types within GDB as well as provide the added information to a GDB user when printing type expressions.

`int` **gdbarch_address_class_type_flags** (`struct gdbarch *gdbarch, int byte_size, int dwarf2_addr_class`) [Function]

Returns the type flags needed to construct a pointer type whose size is *byte_size* and whose address class is *dwarf2_addr_class*. This function is normally called from within a symbol reader. See '`dwarf2read.c`'.

`char *` **gdbarch_address_class_type_flags_to_name** (`struct gdbarch *gdbarch, int type_flags`) [Function]

Given the type flags representing an address class qualifier, return its name.

`int` **gdbarch_address_class_name_to_type_flags** (`struct gdbarch *gdbarch, int name, int *type_flags_ptr`) [Function]

Given an address qualifier name, set the `int` referenced by *type_flags_ptr* to the type flags for that address class qualifier.

Since the need for address classes is rather rare, none of the address class functions are defined by default. Predicate functions are provided to detect when they are defined.

Consider a hypothetical architecture in which addresses are normally 32-bits wide, but 16-bit addresses are also supported. Furthermore, suppose that the DWARF 2 information for this architecture simply uses a `DW_AT_byte_size` value of 2 to indicate the use of one of these "short" pointers. The following functions could be defined to implement the address class functions:

```
somearch_address_class_type_flags (int byte_size,
                                   int dwarf2_addr_class)
{
  if (byte_size == 2)
    return TYPE_FLAG_ADDRESS_CLASS_1;
  else
    return 0;
}

static char *
somearch_address_class_type_flags_to_name (int type_flags)
{
  if (type_flags & TYPE_FLAG_ADDRESS_CLASS_1)
    return "short";
  else
    return NULL;
}

int
somearch_address_class_name_to_type_flags (char *name,
                                           int *type_flags_ptr)
{
  if (strcmp (name, "short") == 0)
    {
      *type_flags_ptr = TYPE_FLAG_ADDRESS_CLASS_1;
      return 1;
    }
  else
    return 0;
}
```

The qualifier `@short` is used in GDB's type expressions to indicate the presence of one of these "short" pointers. For example if the debug information indicates that `short_ptr_var` is one of these short pointers, GDB might show the following behavior:

```
(gdb) ptype short_ptr_var
type = int * @short
```

11.6 Register Representation

11.6.1 Raw and Cooked Registers

GDB considers registers to be a set with members numbered linearly from 0 upwards. The first part of that set corresponds to real physical registers, the second part to any *pseudo-registers*. Pseudo-registers have no independent physical existence, but are useful representations of information within the architecture. For example the OpenRISC 1000 architecture has up to 32 general purpose registers, which are typically represented as 32-bit (or 64-bit) integers. However the GPRs are also used as operands to the floating point operations, and it could be convenient to define a set of pseudo-registers, to show the GPRs represented as floating point values.

For any architecture, the implementer will decide on a mapping from hardware to GDB register numbers. The registers corresponding to real hardware are referred to as *raw*

registers, the remaining registers are *pseudo-registers*. The total register set (raw and pseudo) is called the *cooked* register set.

11.6.2 Functions and Variables Specifying the Register Architecture

These `struct gdbarch` functions and variables specify the number and type of registers in the architecture.

`CORE_ADDR` **read_pc** (`struct regcache *`*regcache*) [Architecture Function]

`void` **write_pc** (`struct regcache *`*regcache*`, CORE_ADDR` *val*) [Architecture Function]

Read or write the program counter. The default value of both functions is `NULL` (no function available). If the program counter is just an ordinary register, it can be specified in `struct gdbarch` instead (see `pc_regnum` below) and it will be read or written using the standard routines to access registers. This function need only be specified if the program counter is not an ordinary register.

Any register information can be obtained using the supplied register cache, *regcache*. See Section 11.6.5 [Register Caching], page 52.

`void` **pseudo_register_read** (`struct gdbarch *`*gdbarch*`, struct regcache *`*regcache*`, int` *regnum*`, const gdb_byte *`*buf*) [Architecture Function]

`void` **pseudo_register_write** (`struct gdbarch *`*gdbarch*`, struct regcache *`*regcache*`, int` *regnum*`, const gdb_byte *`*buf*) [Architecture Function]

These functions should be defined if there are any pseudo-registers. The default value is `NULL`. *regnum* is the number of the register to read or write (which will be a *cooked* register number) and *buf* is the buffer where the value read will be placed, or from which the value to be written will be taken. The value in the buffer may be converted to or from a signed or unsigned integral value using one of the utility functions (see Section 11.6.4 [Using Different Register and Memory Data Representations], page 51).

The access should be for the specified architecture, *gdbarch*. Any register information can be obtained using the supplied register cache, *regcache*. See Section 11.6.5 [Register Caching], page 52.

`int` **sp_regnum** [Architecture Variable]

This specifies the register holding the stack pointer, which may be a raw or pseudo-register. It defaults to -1 (not defined), but it is an error for it not to be defined.

The value of the stack pointer register can be accessed withing GDB as the variable *$sp*.

`int` **pc_regnum** [Architecture Variable]

This specifies the register holding the program counter, which may be a raw or pseudo-register. It defaults to -1 (not defined). If `pc_regnum` is not defined, then the functions `read_pc` and `write_pc` (see above) must be defined.

The value of the program counter (whether defined as a register, or through `read_pc` and `write_pc`) can be accessed withing GDB as the variable *$pc*.

`int` **ps_regnum** [Architecture Variable]

This specifies the register holding the processor status (often called the status register), which may be a raw or pseudo-register. It defaults to -1 (not defined).

If defined, the value of this register can be accessed withing GDB as the variable *$ps*.

`int` **fp0_regnum** [Architecture Variable]

This specifies the first floating point register. It defaults to 0. `fp0_regnum` is not needed unless the target offers support for floating point.

11.6.3 Functions Giving Register Information

These functions return information about registers.

`const char *` **register_name** (`struct gdbarch *`*gdbarch*`, int` *regnum*`)` [Architecture Function]

This function should convert a register number (raw or pseudo) to a register name (as a C `const char *`). This is used both to determine the name of a register for output and to work out the meaning of any register names used as input. The function may also return `NULL`, to indicate that *regnum* is not a valid register.

For example with the OpenRISC 1000, GDB registers 0-31 are the General Purpose Registers, register 32 is the program counter and register 33 is the supervision register (i.e. the processor status register), which map to the strings `"gpr00"` through `"gpr31"`, `"pc"` and `"sr"` respectively. This means that the GDB command *`print $gpr5`* should print the value of the OR1K general purpose register 5[4].

The default value for this function is `NULL`, meaning undefined. It should always be defined.

The access should be for the specified architecture, *gdbarch*.

`struct type *` **register_type** (`struct gdbarch *`*gdbarch*`, int` *regnum*`)` [Architecture Function]

Given a register number, this function identifies the type of data it may be holding, specified as a `struct type`. GDB allows creation of arbitrary types, but a number of built in types are provided (`builtin_type_void`, `builtin_type_int32` etc), together with functions to derive types from these.

Typically the program counter will have a type of "pointer to function" (it points to code), the frame pointer and stack pointer will have types of "pointer to void" (they point to data on the stack) and all other integer registers will have a type of 32-bit integer or 64-bit integer.

This information guides the formatting when displaying register information. The default value is `NULL` meaning no information is available to guide formatting when displaying registers.

[4] Historically, GDB always had a concept of a frame pointer register, which could be accessed via the GDB variable, *$fp*. That concept is now deprecated, recognizing that not all architectures have a frame pointer. However if an architecture does have a frame pointer register, and defines a register or pseudo-register with the name `"fp"`, then that register will be used as the value of the *$fp* variable.

`void` **print_registers_info** (`struct gdbarch *gdbarch, struct ui_file *file, struct frame_info *frame, int regnum, int all`) [Architecture Function]

Define this function to print out one or all of the registers for the GDB *info registers* command. The default value is the function `default_print_registers_info`, which uses the register type information (see `register_type` above) to determine how each register should be printed. Define a custom version of this function for fuller control over how the registers are displayed.

The access should be for the specified architecture, *gdbarch*, with output to the the file specified by the User Interface Independent Output file handle, *file* (see [UI-Independent Output—the ui_out Functions], page 15).

The registers should show their values in the frame specified by *frame*. If *regnum* is -1 and *all* is zero, then all the "significant" registers should be shown (the implementer should decide which registers are "significant"). Otherwise only the value of the register specified by *regnum* should be output. If *regnum* is -1 and *all* is non-zero (true), then the value of all registers should be shown.

By default `default_print_registers_info` prints one register per line, and if *all* is zero omits floating-point registers.

`void` **print_float_info** (`struct gdbarch *gdbarch, struct ui_file *file, struct frame_info *frame, const char *args`) [Architecture Function]

Define this function to provide output about the floating point unit and registers for the GDB *info float* command respectively. The default value is `NULL` (not defined), meaning no information will be provided.

The *gdbarch* and *file* and *frame* arguments have the same meaning as in the `print_registers_info` function above. The string *args* contains any supplementary arguments to the *info float* command.

Define this function if the target supports floating point operations.

`void` **print_vector_info** (`struct gdbarch *gdbarch, struct ui_file *file, struct frame_info *frame, const char *args`) [Architecture Function]

Define this function to provide output about the vector unit and registers for the GDB *info vector* command respectively. The default value is `NULL` (not defined), meaning no information will be provided.

The *gdbarch*, *file* and *frame* arguments have the same meaning as in the `print_registers_info` function above. The string *args* contains any supplementary arguments to the *info vector* command.

Define this function if the target supports vector operations.

`int` **register_reggroup_p** (`struct gdbarch *gdbarch, int regnum, struct reggroup *group`) [Architecture Function]

GDB groups registers into different categories (general, vector, floating point etc). This function, given a register, *regnum*, and group, *group*, returns 1 (true) if the register is in the group and 0 (false) otherwise.

The information should be for the specified architecture, *gdbarch*

The default value is the function `default_register_reggroup_p` which will do a reasonable job based on the type of the register (see the function `register_type` above),

with groups for general purpose registers, floating point registers, vector registers and raw (i.e not pseudo) registers.

11.6.4 Using Different Register and Memory Data Representations

Some architectures have different representations of data objects, depending whether the object is held in a register or memory. 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. An architecture may define the following `struct gdbarch` functions to request conversions between the register and memory representations of a data type:

`int` **gdbarch_convert_register_p** (`struct gdbarch *gdbarch, int reg`) [Architecture Function]

Return non-zero (true) 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. The default value is `NULL` (undefined).

If this function is defined and returns non-zero, the `struct gdbarch` functions `gdbarch_register_to_value` and `gdbarch_value_to_register` (see below) should be used to perform any necessary conversion.

If defined, this function should return zero for the register's native type, when no conversion is necessary.

`void` **gdbarch_register_to_value** (`struct gdbarch *gdbarch, int reg, struct type *type, char *from, char *to`) [Architecture Function]

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:* `gdbarch_register_to_value` and `gdbarch_value_to_register` take their *reg* and *type* arguments in different orders.

`gdbarch_register_to_value` should only be used with registers for which the `gdbarch_convert_register_p` function returns a non-zero value.

`void` **gdbarch_value_to_register** (`struct gdbarch *gdbarch, struct type *type, int reg, char *from, char *to`) [Architecture Function]

Convert a data value of type *type* to register number *reg*' raw format.

> *Note:* `gdbarch_register_to_value` and `gdbarch_value_to_register` take their *reg* and *type* arguments in different orders.

`gdbarch_value_to_register` should only be used with registers for which the `gdbarch_convert_register_p` function returns a non-zero value.

11.6.5 Register Caching

Caching of registers is used, so that the target does not need to be accessed and reanalyzed multiple times for each register in circumstances where the register value cannot have changed.

GDB provides `struct regcache`, associated with a particular `struct gdbarch` to hold the cached values of the raw registers. A set of functions is provided to access both the raw registers (with `raw` in their name) and the full set of cooked registers (with `cooked` in their name). Functions are provided to ensure the register cache is kept synchronized with the values of the actual registers in the target.

Accessing registers through the `struct regcache` routines will ensure that the appropriate `struct gdbarch` functions are called when necessary to access the underlying target architecture. In general users should use the *cooked* functions, since these will map to the *raw* functions automatically as appropriate.

The two key functions are `regcache_cooked_read` and `regcache_cooked_write` which read or write a register from or to a byte buffer (type `gdb_byte *`). For convenience the wrapper functions `regcache_cooked_read_signed`, `regcache_cooked_read_unsigned`, `regcache_cooked_write_signed` and `regcache_cooked_write_unsigned` are provided, which read or write the value using the buffer and convert to or from an integral value as appropriate.

11.7 Frame Interpretation

11.7.1 All About Stack Frames

GDB needs to understand the stack on which local (automatic) variables are stored. The area of the stack containing all the local variables for a function invocation is known as the *stack frame* for that function (or colloquially just as the *frame*). In turn the function that called the function will have its stack frame, and so on back through the chain of functions that have been called.

Almost all architectures have one register dedicated to point to the end of the stack (the *stack pointer*). Many have a second register which points to the start of the currently active stack frame (the *frame pointer*). The specific arrangements for an architecture are a key part of the ABI.

A diagram helps to explain this. Here is a simple program to compute factorials:

```
#include <stdio.h>
int fact (int n)
{
  if (0 == n)
    {
      return 1;
    }
  else
    {
      return n * fact (n - 1);
    }
```

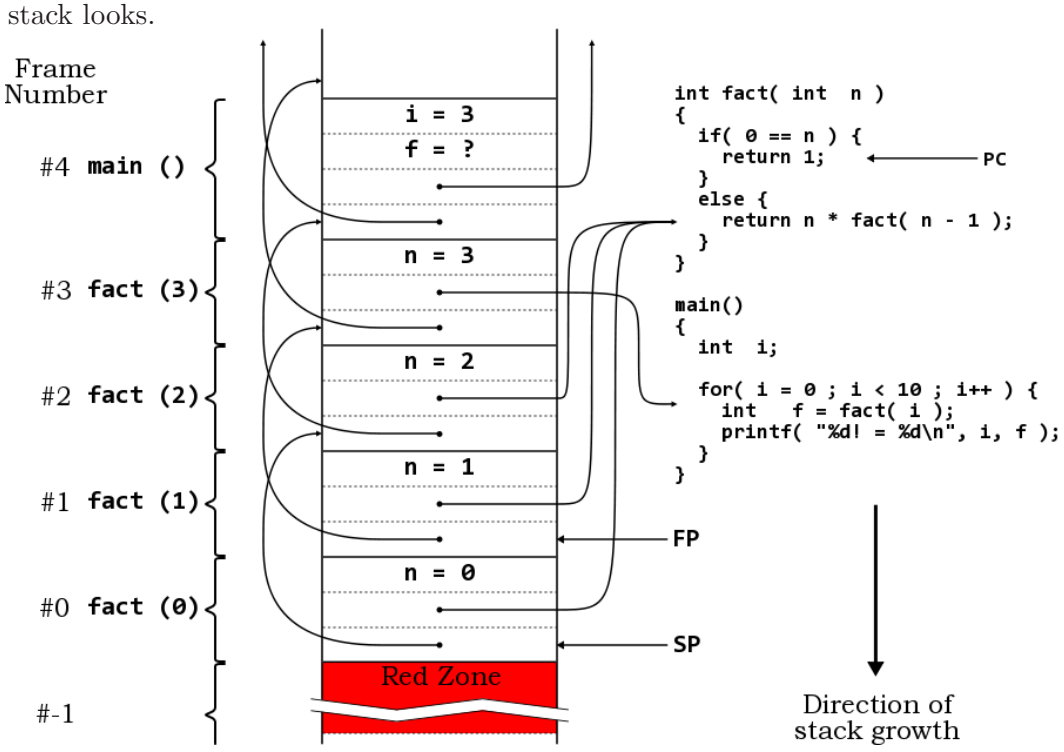
```
}

main ()
{
  int i;

  for (i = 0; i < 10; i++)
    {
      int   f = fact (i);
      printf ("%d! = %d\n", i, f);
    }
}
```

Consider the state of the stack when the code reaches line 6 after the main program has called `fact (3)`. The chain of function calls will be `main ()`, `fact (3)`, `fact (2)`, `fact (1)` and `fact (0)`.

In this illustration the stack is falling (as used for example by the OpenRISC 1000 ABI). The stack pointer (SP) is at the end of the stack (lowest address) and the frame pointer (FP) is at the highest address in the current stack frame. The following diagram shows how the stack looks.



In each stack frame, offset 0 from the stack pointer is the frame pointer of the previous frame and offset 4 (this is illustrating a 32-bit architecture) from the stack pointer is the return address. Local variables are indexed from the frame pointer, with negative indexes. In the function `fact`, offset -4 from the frame pointer is the argument *n*. In the `main` function, offset -4 from the frame pointer is the local variable *i* and offset -8 from the frame pointer is the local variable *f*[5].

[5] This is a simplified example for illustrative purposes only. Good optimizing compilers would not put anything on the stack for such simple functions. Indeed they might eliminate the recursion and use of the stack entirely!

It is very easy to get confused when examining stacks. GDB has terminology it uses rigorously throughout. The stack frame of the function currently executing, or where execution stopped is numbered zero. In this example frame #0 is the stack frame of the call to `fact (0)`. The stack frame of its calling function (`fact (1)` in this case) is numbered #1 and so on back through the chain of calls.

The main GDB data structure describing frames is `struct frame_info`. It is not used directly, but only via its accessor functions. `frame_info` includes information about the registers in the frame and a pointer to the code of the function with which the frame is associated. The entire stack is represented as a linked list of `frame_info` structs.

11.7.2 Frame Handling Terminology

It is easy to get confused when referencing stack frames. GDB uses some precise terminology.

- *THIS* frame is the frame currently under consideration.
- The *NEXT* frame, also sometimes called the inner or newer frame is the frame of the function called by the function of THIS frame.
- The *PREVIOUS* frame, also sometimes called the outer or older frame is the frame of the function which called the function of THIS frame.

So in the example in the previous section (see Section 11.7.1 [All About Stack Frames], page 52), if THIS frame is #3 (the call to `fact (3)`), the NEXT frame is frame #2 (the call to `fact (2)`) and the PREVIOUS frame is frame #4 (the call to `main ()`).

The *innermost* frame is the frame of the current executing function, or where the program stopped, in this example, in the middle of the call to `fact (0)`). It is always numbered frame #0.

The *base* of a frame is the address immediately before the start of the NEXT frame. For a stack which grows down in memory (a *falling* stack) this will be the lowest address and for a stack which grows up in memory (a *rising* stack) this will be the highest address in the frame.

GDB functions to analyze the stack are typically given a pointer to the NEXT frame to determine information about THIS frame. Information about THIS frame includes data on where the registers of the PREVIOUS frame are stored in this stack frame. In this example the frame pointer of the PREVIOUS frame is stored at offset 0 from the stack pointer of THIS frame.

The process whereby a function is given a pointer to the NEXT frame to work out information about THIS frame is referred to as *unwinding*. The GDB functions involved in this typically include unwind in their name.

The process of analyzing a target to determine the information that should go in struct frame_info is called *sniffing*. The functions that carry this out are called sniffers and typically include sniffer in their name. More than one sniffer may be required to extract all the information for a particular frame.

Because so many functions work using the NEXT frame, there is an issue about addressing the innermost frame—it has no NEXT frame. To solve this GDB creates a dummy frame #-1, known as the *sentinel* frame.

11.7.3 Prologue Caches

All the frame sniffing functions typically examine the code at the start of the corresponding function, to determine the state of registers. The ABI will save old values and set new values of key registers at the start of each function in what is known as the function *prologue*.

For any particular stack frame this data does not change, so all the standard unwinding functions, in addition to receiving a pointer to the NEXT frame as their first argument, receive a pointer to a *prologue cache* as their second argument. This can be used to store values associated with a particular frame, for reuse on subsequent calls involving the same frame.

It is up to the user to define the structure used (it is a `void *` pointer) and arrange allocation and deallocation of storage. However for general use, GDB provides `struct trad_frame_cache`, with a set of accessor routines. This structure holds the stack and code address of THIS frame, the base address of the frame, a pointer to the struct `frame_info` for the NEXT frame and details of where the registers of the PREVIOUS frame may be found in THIS frame.

Typically the first time any sniffer function is called with NEXT frame, the prologue sniffer for THIS frame will be `NULL`. The sniffer will analyze the frame, allocate a prologue cache structure and populate it. Subsequent calls using the same NEXT frame will pass in this prologue cache, so the data can be returned with no additional analysis.

11.7.4 Functions and Variable to Analyze Frames

These struct `gdbarch` functions and variable should be defined to provide analysis of the stack frame and allow it to be adjusted as required.

`CORE_ADDR` **skip_prologue** (`struct gdbarch *`*`gdbarch`*`, CORE_ADDR` *`pc`*`)` [Architecture Function]

The prologue of a function is the code at the beginning of the function which sets up the stack frame, saves the return address etc. The code representing the behavior of the function starts after the prologue.

This function skips past the prologue of a function if the program counter, *pc*, is within the prologue of a function. The result is the program counter immediately after the prologue. With modern optimizing compilers, this may be a far from trivial exercise. However the required information may be within the binary as DWARF2 debugging information, making the job much easier.

The default value is `NULL` (not defined). This function should always be provided, but can take advantage of DWARF2 debugging information, if that is available.

`int` **inner_than** (`CORE_ADDR` *`lhs`*`, CORE_ADDR` *`rhs`*`)` [Architecture Function]

Given two frame or stack pointers, return non-zero (true) if the first represents the *inner* stack frame and 0 (false) otherwise. This is used to determine whether the target has a stack which grows up in memory (rising stack) or grows down in memory (falling stack). See Section 11.7.1 [All About Stack Frames], page 52, for an explanation of *inner* frames.

The default value of this function is `NULL` and it should always be defined. However for almost all architectures one of the built-in functions can be used: `core_addr_lessthan` (for stacks growing down in memory) or `core_addr_greaterthan` (for stacks growing up in memory).

`CORE_ADDR` **frame_align** (`struct gdbarch *`*gdbarch*, `CORE_ADDR` *address*) [Architecture Function]

The architecture may have constraints on how its frames are aligned. For example the OpenRISC 1000 ABI requires stack frames to be double-word aligned, but 32-bit versions of the architecture allocate single-word values to the stack. Thus extra padding may be needed at the end of a stack frame.

Given a proposed address for the stack pointer, this function returns a suitably aligned address (by expanding the stack frame).

The default value is `NULL` (undefined). This function should be defined for any architecture where it is possible the stack could become misaligned. The utility functions `align_down` (for falling stacks) and `align_up` (for rising stacks) will facilitate the implementation of this function.

`int` **frame_red_zone_size** [Architecture Variable]

Some ABIs reserve space beyond the end of the stack for use by leaf functions without prologue or epilogue or by exception handlers (for example the OpenRISC 1000).

This is known as a *red zone* (AMD terminology). The AMD64 (nee x86-64) ABI documentation refers to the *red zone* when describing this scratch area.

The default value is 0. Set this field if the architecture has such a red zone. The value must be aligned as required by the ABI (see `frame_align` above for an explanation of stack frame alignment).

11.7.5 Functions to Access Frame Data

These functions provide access to key registers and arguments in the stack frame.

`CORE_ADDR` **unwind_pc** (`struct gdbarch *`*gdbarch*, `struct frame_info *`*next_frame*) [Architecture Function]

This function is given a pointer to the NEXT stack frame (see Section 11.7.1 [All About Stack Frames], page 52, for how frames are represented) and returns the value of the program counter in the PREVIOUS frame (i.e. the frame of the function that called THIS one). This is commonly referred to as the *return address*.

The implementation, which must be frame agnostic (work with any frame), is typically no more than:

```
ULONGEST pc;
pc = frame_unwind_register_unsigned (next_frame, ARCH_PC_REGNUM);
return gdbarch_addr_bits_remove (gdbarch, pc);
```

`CORE_ADDR` **unwind_sp** (`struct gdbarch *`*gdbarch*, `struct frame_info *`*next_frame*) [Architecture Function]

This function is given a pointer to the NEXT stack frame (see Section 11.7.1 [All About Stack Frames], page 52 for how frames are represented) and returns the value

of the stack pointer in the PREVIOUS frame (i.e. the frame of the function that called THIS one).

The implementation, which must be frame agnostic (work with any frame), is typically no more than:

```
ULONGEST sp;
sp = frame_unwind_register_unsigned (next_frame, ARCH_SP_REGNUM);
return gdbarch_addr_bits_remove (gdbarch, sp);
```

`int` **frame_num_args** (`struct gdbarch *`*gdbarch*`, struct frame_info *`*this_frame*) [Architecture Function]

This function is given a pointer to THIS stack frame (see Section 11.7.1 [All About Stack Frames], page 52 for how frames are represented), and returns the number of arguments that are being passed, or -1 if not known.

The default value is `NULL` (undefined), in which case the number of arguments passed on any stack frame is always unknown. For many architectures this will be a suitable default.

11.7.6 Analyzing Stacks—Frame Sniffers

When a program stops, GDB needs to construct the chain of struct `frame_info` representing the state of the stack using appropriate *sniffers*.

Each architecture requires appropriate sniffers, but they do not form entries in `struct gdbarch`, since more than one sniffer may be required and a sniffer may be suitable for more than one `struct gdbarch`. Instead sniffers are associated with architectures using the following functions.

- `frame_unwind_append_sniffer` is used to add a new sniffer to analyze THIS frame when given a pointer to the NEXT frame.
- `frame_base_append_sniffer` is used to add a new sniffer which can determine information about the base of a stack frame.
- `frame_base_set_default` is used to specify the default base sniffer.

These functions all take a reference to `struct gdbarch`, so they are associated with a specific architecture. They are usually called in the `gdbarch` initialization function, after the `gdbarch` struct has been set up. Unless a default has been set, the most recently appended sniffer will be tried first.

The main frame unwinding sniffer (as set by `frame_unwind_append_sniffer)` returns a structure specifying a set of sniffing functions:

```
struct frame_unwind
{
   enum frame_type            type;
   frame_this_id_ftype       *this_id;
   frame_prev_register_ftype *prev_register;
   const struct frame_data   *unwind_data;
   frame_sniffer_ftype       *sniffer;
   frame_prev_pc_ftype       *prev_pc;
   frame_dealloc_cache_ftype *dealloc_cache;
};
```

The `type` field indicates the type of frame this sniffer can handle: normal, dummy (see Section 11.8.2 [Functions Creating Dummy Frames], page 59), signal handler or sentinel. Signal handlers sometimes have their own simplified stack structure for efficiency, so may need their own handlers.

The `unwind_data` field holds additional information which may be relevant to particular types of frame. For example it may hold additional information for signal handler frames.

The remaining fields define functions that yield different types of information when given a pointer to the NEXT stack frame. Not all functions need be provided. If an entry is `NULL`, the next sniffer will be tried instead.

- `this_id` determines the stack pointer and function (code entry point) for THIS stack frame.
- `prev_register` determines where the values of registers for the PREVIOUS stack frame are stored in THIS stack frame.
- `sniffer` takes a look at THIS frame's registers to determine if this is the appropriate unwinder.
- `prev_pc` determines the program counter for THIS frame. Only needed if the program counter is not an ordinary register (see Section 11.6.2 [Functions and Variables Specifying the Register Architecture], page 48).
- `dealloc_cache` frees any additional memory associated with the prologue cache for this frame (see Section 11.7.3 [Prologue Caches], page 55).

In general it is only the `this_id` and `prev_register` fields that need be defined for custom sniffers.

The frame base sniffer is much simpler. It is a `struct frame_base`, which refers to the corresponding `frame_unwind` struct and whose fields refer to functions yielding various addresses within the frame.

```
struct frame_base
{
   const struct frame_unwind *unwind;
   frame_this_base_ftype     *this_base;
   frame_this_locals_ftype   *this_locals;
   frame_this_args_ftype     *this_args;
};
```

All the functions referred to take a pointer to the NEXT frame as argument. The function referred to by `this_base` returns the base address of THIS frame, the function referred to by `this_locals` returns the base address of local variables in THIS frame and the function referred to by `this_args` returns the base address of the function arguments in this frame.

As described above, the base address of a frame is the address immediately before the start of the NEXT frame. For a falling stack, this is the lowest address in the frame and for a rising stack it is the highest address in the frame. For most architectures the same address is also the base address for local variables and arguments, in which case the same function can be used for all three entries[6].

[6] It is worth noting that if it cannot be determined in any other way (for example by there being a register with the name `"fp"`), then the result of the `this_base` function will be used as the value of the frame pointer variable *$fp* in GDB. This is very often not correct (for example with the OpenRISC 1000, this

11.8 Inferior Call Setup

11.8.1 About Dummy Frames

GDB can call functions in the target code (for example by using the *call* or *print* commands). These functions may be breakpointed, and it is essential that if a function does hit a breakpoint, commands like *backtrace* work correctly.

This is achieved by making the stack look as though the function had been called from the point where GDB had previously stopped. This requires that GDB can set up stack frames appropriate for such function calls.

11.8.2 Functions Creating Dummy Frames

The following functions provide the functionality to set up such *dummy* stack frames.

`CORE_ADDR` **push_dummy_call** (`struct gdbarch *`*gdbarch*, `struct value *`*function*, `struct regcache *`*regcache*, `CORE_ADDR` *bp_addr*, `int` *nargs*, `struct value **`*args*, `CORE_ADDR` *sp*, `int` *struct_return*, `CORE_ADDR` *struct_addr*) [Architecture Function]

This function sets up a dummy stack frame for the function about to be called. `push_dummy_call` is given the arguments to be passed and must copy them into registers or push them on to the stack as appropriate for the ABI.

function is a pointer to the function that will be called and *regcache* the register cache from which values should be obtained. *bp_addr* is the address to which the function should return (which is breakpointed, so GDB can regain control, hence the name). *nargs* is the number of arguments to pass and *args* an array containing the argument values. *struct_return* is non-zero (true) if the function returns a structure, and if so *struct_addr* is the address in which the structure should be returned.

After calling this function, GDB will pass control to the target at the address of the function, which will find the stack and registers set up just as expected.

The default value of this function is `NULL` (undefined). If the function is not defined, then GDB will not allow the user to call functions within the target being debugged.

`struct frame_id` **unwind_dummy_id** (`struct gdbarch *`*gdbarch*, `struct frame_info *`*next_frame*) [Architecture Function]

This is the inverse of `push_dummy_call` which restores the stack pointer and program counter after a call to evaluate a function using a dummy stack frame. The result is a `struct frame_id`, which contains the value of the stack pointer and program counter to be used.

The NEXT frame pointer is provided as argument, *next_frame*. THIS frame is the frame of the dummy function, which can be unwound, to yield the required stack pointer and program counter from the PREVIOUS frame.

---

value is the stack pointer, *$sp*). In this case a register (raw or pseudo) with the name `"fp"` should be defined. It will be used in preference as the value of *$fp*.

The default value is `NULL` (undefined). If `push_dummy_call` is defined, then this function should also be defined.

`CORE_ADDR` **push_dummy_code** (`struct gdbarch *`*gdbarch*`, CORE_ADDR` *sp*`, CORE_ADDR` *funaddr*`, struct value **`*args*`, int` *nargs*`, struct type *`*value_type*`, CORE_ADDR *`*real_pc*`, CORE_ADDR *`*bp_addr*`, struct regcache *`*regcache*`)` [Architecture Function]

If this function is not defined (its default value is `NULL`), a dummy call will use the entry point of the currently loaded code on the target as its return address. A temporary breakpoint will be set there, so the location must be writable and have room for a breakpoint.

It is possible that this default is not suitable. It might not be writable (in ROM possibly), or the ABI might require code to be executed on return from a call to unwind the stack before the breakpoint is encountered.

If either of these is the case, then push_dummy_code should be defined to push an instruction sequence onto the end of the stack to which the dummy call should return.

The arguments are essentially the same as those to `push_dummy_call`. However the function is provided with the type of the function result, *value_type*, *bp_addr* is used to return a value (the address at which the breakpoint instruction should be inserted) and *real pc* is used to specify the resume address when starting the call sequence. The function should return the updated innermost stack address.

> *Note:* This does require that code in the stack can be executed. Some Harvard architectures may not allow this.

11.9 Adding support for debugging core files

The prerequisite for adding core file support in GDB is to have core file support in BFD.

Once BFD support is available, writing the apropriate `regset_from_core_section` architecture function should be all that is needed in order to add support for core files in GDB.

11.10 Defining Other Architecture Features

This section describes other functions and values in `gdbarch`, together with some useful macros, that you can use to define the target architecture.

`CORE_ADDR gdbarch_addr_bits_remove (`*gdbarch*`, `*addr*`)`

If a raw machine instruction address includes any bits that are not really part of the address, then this function is used to zero 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. `gdbarch_addr_bits_remove` would then for example look like that:

```
arch_addr_bits_remove (CORE_ADDR addr)
{
  return (addr &= ~0x3);
}
```

`int address_class_name_to_type_flags (`*`gdbarch`*`, `*`name`*`, `*`type_flags_ptr`*`)`

If *name* is a valid address class qualifier name, set the `int` referenced by *type_flags_ptr* to the mask representing the qualifier and return 1. If *name* is not a valid address class qualifier name, return 0.

The value for *type_flags_ptr* should be one of `TYPE_FLAG_ADDRESS_CLASS_1`, `TYPE_FLAG_ADDRESS_CLASS_2`, or possibly some combination of these values or'd together. See Chapter 11 [Address Classes], page 39.

`int address_class_name_to_type_flags_p (`*`gdbarch`*`)`

Predicate which indicates whether `address_class_name_to_type_flags` has been defined.

`int gdbarch_address_class_type_flags (`*`gdbarch`*`, `*`byte_size`*`, `*`dwarf2_addr_class`*`)`

Given a pointers byte size (as described by the debug information) and the possible `DW_AT_address_class` value, return the type flags used by GDB to represent this address class. The value returned should be one of `TYPE_FLAG_ADDRESS_CLASS_1`, `TYPE_FLAG_ADDRESS_CLASS_2`, or possibly some combination of these values or'd together. See Chapter 11 [Address Classes], page 39.

`int gdbarch_address_class_type_flags_p (`*`gdbarch`*`)`

Predicate which indicates whether `gdbarch_address_class_type_flags_p` has been defined.

`const char *gdbarch_address_class_type_flags_to_name (`*`gdbarch`*`, `*`type_flags`*`)`

Return the name of the address class qualifier associated with the type flags given by *type_flags*.

`int gdbarch_address_class_type_flags_to_name_p (`*`gdbarch`*`)`

Predicate which indicates whether `gdbarch_address_class_type_flags_to_name` has been defined. See Chapter 11 [Address Classes], page 39.

`void gdbarch_address_to_pointer (`*`gdbarch`*`, `*`type`*`, `*`buf`*`, `*`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. See Chapter 11 [Pointers Are Not Always Addresses], page 39.

`int gdbarch_believe_pcc_promotion (`*`gdbarch`*`)`

Used to notify 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.

`gdbarch_bits_big_endian (`*`gdbarch`*`)`

This is used if the numbering of bits in the targets does **not** match the endianism 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.

`set_gdbarch_bits_big_endian (gdbarch, bits_big_endian)`
: Calling set_gdbarch_bits_big_endian with a value of 1 indicates that the bits in the target are numbered in a big-endian bit order, 0 indicates 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 `gdbarch_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 `gdbarch_breakpoint_from_pc`.

`const gdb_byte *gdbarch_breakpoint_from_pc (gdbarch, pcptr, lenptr)`
: Use the program counter to determine the contents and size of a breakpoint instruction. It returns a pointer to a static string of bytes that encode a breakpoint instruction, stores the length of the string to `*lenptr`, and adjusts the program counter (if necessary) to point to the actual memory location where the breakpoint should be inserted. May return `NULL` to indicate that software breakpoints are not supported.

  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.

  Provided breakpoint bytes can be also used by `bp_loc_is_permanent` to detect permanent breakpoints. `gdbarch_breakpoint_from_pc` should return an unchanged memory copy if it was called for a location with permanent breakpoint as some architectures use breakpoint instructions containing arbitrary parameter value.

  Replaces all the other *BREAKPOINT* macros.

`int gdbarch_memory_insert_breakpoint (gdbarch, bp_tgt)`
`gdbarch_memory_remove_breakpoint (gdbarch, bp_tgt)`
: 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 set these for most architectures. Architectures which may want to set `gdbarch_memory_insert_breakpoint` and `gdbarch_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 `gdbarch_breakpoint_from_pc` needs to read the target's memory for some reason.

`CORE_ADDR gdbarch_adjust_breakpoint_address (gdbarch, bpaddr)`

Given an address at which a breakpoint is desired, return a breakpoint address adjusted to account for architectural constraints on breakpoint placement. This method is not needed by most targets.

The FR-V target (see '`frv-tdep.c`') requires this method. The FR-V is a VLIW architecture in which a number of RISC-like instructions are grouped (packed) together into an aggregate instruction or instruction bundle. When the processor executes one of these bundles, the component instructions are executed in parallel.

In the course of optimization, the compiler may group instructions from distinct source statements into the same bundle. The line number information associated with one of the latter statements will likely refer to some instruction other than the first one in the bundle. So, if the user attempts to place a breakpoint on one of these latter statements, GDB must be careful to *not* place the break instruction on any instruction other than the first one in the bundle. (Remember though that the instructions within a bundle execute in parallel, so the *first* instruction is the instruction at the lowest address and has nothing to do with execution order.)

The FR-V's `gdbarch_adjust_breakpoint_address` method will adjust a breakpoint's address by scanning backwards for the beginning of the bundle, returning the address of the bundle.

Since the adjustment of a breakpoint may significantly alter a user's expectation, GDB prints a warning when an adjusted breakpoint is initially set and each time that that breakpoint is hit.

`int gdbarch_call_dummy_location (gdbarch)`

See the file '`inferior.h`'.

This method has been replaced by `gdbarch_push_dummy_code` (see [gdbarch_push_dummy_code], page 66).

`int gdbarch_cannot_fetch_register (gdbarch, regum)`

This function should return nonzero if *regno* cannot be fetched from an inferior process.

`int gdbarch_cannot_store_register (gdbarch, regnum)`

This function should return nonzero if *regno* should not be written to the target. This is often the case for program counters, status words, and other special registers. This function returns 0 as default so that GDB will assume that all registers may be written.

`int gdbarch_convert_register_p (gdbarch, regnum, struct type *type)`

Return non-zero if register *regnum* represents data values of type *type* in a non-standard form. See Chapter 11 [Using Different Register and Memory Data Representations], page 39.

`int gdbarch_fp0_regnum (gdbarch)`

This function returns the number of the first floating point register, if the machine has such registers. Otherwise, it returns -1.

`CORE_ADDR gdbarch_decr_pc_after_break (gdbarch)`
: This function shall return 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.

`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.

`int gdbarch_dwarf2_reg_to_regnum (gdbarch, dwarf2_regnr)`
: Convert DWARF2 register number *dwarf2_regnr* into GDB regnum. If not defined, no conversion will be performed.

`int gdbarch_ecoff_reg_to_regnum (gdbarch, ecoff_regnr)`
: Convert ECOFF register number *ecoff_regnr* into GDB regnum. If not defined, no conversion will be performed.

`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.)

`gdbarch_get_longjmp_target`
: This function 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, typically by using a manually-determined offset into the `jmp_buf`. (While we might like to get the offset from the target's '`jmpbuf.h`', that header file cannot be assumed to be available when building a cross-debugger.)

`DEPRECATED_IBM6000_TARGET`
: Shows that we are configured for an IBM RS/6000 system. This conditional should be eliminated (FIXME) and replaced by feature-specific macros. It was introduced in 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 4.

`gdbarch_in_function_epilogue_p (gdbarch, addr)`
: Returns non-zero if the given *addr* 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.

`int gdbarch_in_solib_return_trampoline (gdbarch, pc, name)`
: Define this function to return nonzero if the program is stopped in the trampoline that returns from a shared library.

`target_so_ops.in_dynsym_resolve_code (pc)`
: Define this to return 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.

`CORE_ADDR gdbarch_integer_to_address (gdbarch, type, buf)`
: Define this when the architecture needs to handle non-pointer to address conversions 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* `gdbarch_integer_to_address` *certainly makes it possible for* GDB *to "get it right" in all circumstances.*

  See Chapter 11 [Pointers Are Not Always Addresses], page 39.

`CORE_ADDR gdbarch_pointer_to_address (gdbarch, 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 11 [Pointers Are Not Always Addresses], page 39.

`void gdbarch_register_to_value(gdbarch, frame, regnum, type, fur)`
: Convert the raw contents of register *regnum* into a value of type *type*. See Chapter 11 [Using Different Register and Memory Data Representations], page 39.

`REGISTER_CONVERT_TO_VIRTUAL(reg, type, from, to)`
: Convert the value of register *reg* from its raw form to its virtual form. See Chapter 11 [Raw and Virtual Register Representations], page 39.

`REGISTER_CONVERT_TO_RAW(type, reg, from, to)`
: Convert the value of register *reg* from its virtual form to its raw form. See Chapter 11 [Raw and Virtual Register Representations], page 39.

`const struct regset *regset_from_core_section (struct gdbarch * gdbarch, const char * sect_name, size_t sect_size)`
: Return the appropriate register set for a core file section with name *sect_name* and size *sect_size*.

`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_breakpoints_p)`
: A function that inserts or removes (depending on *insert_breakpoints_p*) breakpoints at each possible destinations of the next instruction. See '`sparc-tdep.c`' and '`rs6000-tdep.c`' for examples.

`set_gdbarch_sofun_address_maybe_missing (gdbarch, set)`

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.

Calling `set_gdbarch_sofun_address_maybe_missing` with a non-zero argument *set* 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.

In this case, GDB assumes two things:

- `N_FUN` stabs have an address of zero. Instead of using those addresses, you should find the address 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.

`int gdbarch_stabs_argument_has_addr (gdbarch, type)`

Define this function to return nonzero if a function argument of type *type* is passed by reference instead of value.

`CORE_ADDR gdbarch_push_dummy_call (gdbarch, function, regcache, bp_addr, nargs, args, sp, struct_return, struct_addr)`

Define this to push the dummy frame's call to the inferior function onto the stack. In addition to pushing *nargs*, the code should push *struct_addr* (when *struct_return* is non-zero), and the return address (*bp_addr*).

function is a pointer to a `struct value`; on architectures that use function descriptors, this contains the function descriptor value.

Returns the updated top-of-stack pointer.

`CORE_ADDR gdbarch_push_dummy_code (gdbarch, sp, funaddr, using_gcc, args, nargs, value_type, real_pc, bp_addr, regcache)`

Given a stack based call dummy, push the instruction sequence (including space for a breakpoint) to which the called function should return.

Set *bp_addr* to the address at which the breakpoint instruction should be inserted, *real_pc* to the resume address when starting the call sequence, and return the updated inner-most stack address.

By default, the stack is grown sufficient to hold a frame-aligned (see [frame_align], page 56) breakpoint, *bp_addr* is set to the address reserved for that breakpoint, and *real_pc* set to *funaddr*.

This method replaces `gdbarch_call_dummy_location (gdbarch)`.

`int gdbarch_sdb_reg_to_regnum (gdbarch, sdb_regnr)`

Use this function to convert sdb register *sdb_regnr* into GDB regnum. If not defined, no conversion will be done.

`enum return_value_convention gdbarch_return_value (struct gdbarch *gdbarch, struct type *valtype, struct regcache *regcache, void *readbuf, const void *writebuf)`

Given a function with a return-value of type *rettype*, return which return-value convention that function would use.

GDB currently recognizes two function return-value conventions: `RETURN_VALUE_REGISTER_CONVENTION` where the return value is found in registers; and `RETURN_VALUE_STRUCT_CONVENTION` where the return value is found in memory and the address of that memory location is passed in as the function's first parameter.

If the register convention is being used, and *writebuf* is non-`NULL`, also copy the return-value in *writebuf* into *regcache*.

If the register convention is being used, and *readbuf* is non-`NULL`, also copy the return value from *regcache* into *readbuf* (*regcache* contains a copy of the registers from the just returned function).

Maintainer note: This method replaces separate predicate, extract, store methods. By having only one method, the logic needed to determine the return-value convention need only be implemented in one place. If GDB *were written in an* OO *language, this method would instead return an object that knew how to perform the register return-value extract and store.*

Maintainer note: This method does not take a `gcc_p` *parameter, and such a parameter should not be added. If an architecture that requires per-compiler or per-function information be identified, then the replacement of* `rettype` *with* `struct value` *function should be pursued.*

Maintainer note: The `regcache` *parameter limits this methods to the inner most frame. While replacing* `regcache` *with a* `struct frame_info` *frame parameter would remove that limitation there has yet to be a demonstrated need for such a change.*

`void gdbarch_skip_permanent_breakpoint (gdbarch, regcache)`

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 `gdbarch_skip_permanent_breakpoint` adjusts the processor's state so that execution will resume just after the breakpoint. This function does the right thing even when the breakpoint is in the delay slot of a branch or jump.

`CORE_ADDR gdbarch_skip_trampoline_code (gdbarch, frame, pc)`

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

`int gdbarch_deprecated_fp_regnum (gdbarch)`
: If the frame pointer is in a register, use this function to return the number of that register.

`int gdbarch_stab_reg_to_regnum (gdbarch, stab_regnr)`
: Use this function to convert stab register *stab_regnr* into GDB regnum. If not defined, no conversion will be done.

`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.

`int gdbarch_char_signed (gdbarch)`
: 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.

`int gdbarch_double_bit (gdbarch)`
: Number of bits in a double float; defaults to `8 * TARGET_CHAR_BIT`.

`int gdbarch_float_bit (gdbarch)`
: Number of bits in a float; defaults to `4 * TARGET_CHAR_BIT`.

`int gdbarch_int_bit (gdbarch)`
: Number of bits in an integer; defaults to `4 * TARGET_CHAR_BIT`.

`int gdbarch_long_bit (gdbarch)`
: Number of bits in a long integer; defaults to `4 * TARGET_CHAR_BIT`.

`int gdbarch_long_double_bit (gdbarch)`
: Number of bits in a long double float; defaults to `2 * gdbarch_double_bit (gdbarch)`.

`int gdbarch_long_long_bit (gdbarch)`
: Number of bits in a long long integer; defaults to `2 * gdbarch_long_bit (gdbarch)`.

`int gdbarch_ptr_bit (gdbarch)`
: Number of bits in a pointer; defaults to `gdbarch_int_bit (gdbarch)`.

`int gdbarch_short_bit (gdbarch)`
: Number of bits in a short integer; defaults to `2 * TARGET_CHAR_BIT`.

`void gdbarch_virtual_frame_pointer (gdbarch, pc, frame_regnum, frame_offset)`
: 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 `gdbarch_deprecated_fp_regnum` (or `gdbarch_sp_regnum`, if no frame pointer is defined), 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 4, for more details and other related macros.

`int gdbarch_print_insn (gdbarch, vma, info)`
: This is the function used by GDB to print an assembly instruction. It prints the instruction at address *vma* in debugged memory and returns the length of the instruction, in bytes. This usually points to a function in the `opcodes` library (see Chapter 15 [Opcodes], page 75). *info* is a structure (of type `disassemble_info`) defined in the header file '`include/dis-asm.h`', and used to pass information to the instruction decoding routine.

`frame_id gdbarch_dummy_id (gdbarch, frame)`
: Given *frame* return a `struct frame_id` that uniquely identifies an inferior function call's dummy frame. The value returned must match the dummy frame stack value previously saved by `call_function_by_hand`.

`void gdbarch_value_to_register (gdbarch, frame, type, buf)`
: Convert a value of type *type* into the raw contents of a register. See Chapter 11 [Using Different Register and Memory Data Representations], page 39.

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`.

11.11 Adding a New Target

The following files add a target to GDB:

'`gdb/ttt-tdep.c`'
: Contains any miscellaneous code required for this target machine. On some machines it doesn't exist at all.

'`gdb/arch-tdep.c`'
'`gdb/arch-tdep.h`'
: This is required to describe the basic layout of the target machine's processor chip (registers, stack, etc.). It can be shared among many targets that use the same processor architecture.

(Target header files such as '`gdb/config/arch/tm-ttt.h`', '`gdb/config/arch/tm-arch.h`', and '`config/tm-os.h`' are no longer used.)

A GDB description for a new architecture, arch is created by defining a global function `_initialize_arch_tdep`, by convention in the source file '`arch-tdep.c`'. For example, in the case of the OpenRISC 1000, this function is called `_initialize_or1k_tdep` and is found in the file '`or1k-tdep.c`'.

The object file resulting from compiling this source file, which will contain the implementation of the `_initialize_arch_tdep` function is specified in the GDB '`configure.tgt`' file, which includes a large case statement pattern matching against the `--target` option of the *`configure`* script.

> *Note:* If the architecture requires multiple source files, the corresponding binaries should be included in '`configure.tgt`'. However if there are header files, the dependencies on these will not be picked up from the entries in '`configure.tgt`'. The '`Makefile.in`' file will need extending to show these dependencies.

A new struct gdbarch, defining the new architecture, is created within the `_initialize_arch_tdep` function by calling `gdbarch_register`:

```
void gdbarch_register (enum bfd_architecture    architecture,
                       gdbarch_init_ftype      *init_func,
                       gdbarch_dump_tdep_ftype *tdep_dump_func);
```

This function has been described fully in an earlier section. See Section 11.2.1 [How an Architecture is Represented], page 42.

The new `struct gdbarch` should contain implementations of the necessary functions (described in the previous sections) to describe the basic layout of the target machine's processor chip (registers, stack, etc.). It can be shared among many targets that use the same processor architecture.

12 Target Descriptions

The target architecture definition (see Chapter 11 [Target Architecture Definition], page 39) contains GDB's hard-coded knowledge about an architecture. For some platforms, it is handy to have more flexible knowledge about a specific instance of the architecture—for instance, a processor or development board. *Target descriptions* provide a mechanism for the user to tell GDB more about what their target supports, or for the target to tell GDB directly.

For details on writing, automatically supplying, and manually selecting target descriptions, see section "Target Descriptions" in *Debugging with* GDB. This section will cover some related topics about the GDB internals.

12.1 Target Descriptions Implementation

Before GDB connects to a new target, or runs a new program on an existing target, it discards any existing target description and reverts to a default gdbarch. Then, after connecting, it looks for a new target description by calling `target_find_description`.

A description may come from a user specified file (XML), the remote '`qXfer:features:read`' packet (also XML), or from any custom `to_read_description` routine in the target vector. For instance, the remote target supports guessing whether a MIPS target is 32-bit or 64-bit based on the size of the '`g`' packet.

If any target description is found, GDB creates a new gdbarch incorporating the description by calling `gdbarch_update_p`. Any '`<architecture>`' element is handled first, to determine which architecture's gdbarch initialization routine is called to create the new architecture. Then the initialization routine is called, and has a chance to adjust the constructed architecture based on the contents of the target description. For instance, it can recognize any properties set by a `to_read_description` routine. Also see Section 12.2 [Adding Target Described Register Support], page 71.

12.2 Adding Target Described Register Support

Target descriptions can report additional registers specific to an instance of the target. But it takes a little work in the architecture specific routines to support this.

A target description must either have no registers or a complete set—this avoids complexity in trying to merge standard registers with the target defined registers. It is the architecture's responsibility to validate that a description with registers has everything it needs. To keep architecture code simple, the same mechanism is used to assign fixed internal register numbers to standard registers.

If `tdesc_has_registers` returns 1, the description contains registers. The architecture's `gdbarch_init` routine should:

- Call `tdesc_data_alloc` to allocate storage, early, before searching for a matching gdbarch or allocating a new one.
- Use `tdesc_find_feature` to locate standard features by name.
- Use `tdesc_numbered_register` and `tdesc_numbered_register_choices` to locate the expected registers in the standard features.
- Return `NULL` if a required feature is missing, or if any standard feature is missing expected registers. This will produce a warning that the description was incomplete.
- Free the allocated data before returning, unless `tdesc_use_registers` is called.
- Call `set_gdbarch_num_regs` as usual, with a number higher than any fixed number passed to `tdesc_numbered_register`.
- Call `tdesc_use_registers` after creating a new gdbarch, before returning it.

After `tdesc_use_registers` has been called, the architecture's `register_name`, `register_type`, and `register_reggroup_p` routines will not be called; that information will be taken from the target description. `num_regs` may be increased to account for any additional registers in the description.

Pseudo-registers require some extra care:

- Using `tdesc_numbered_register` allows the architecture to give constant register numbers to standard architectural registers, e.g. as an `enum` in '*arch*`-tdep.h`'. But because pseudo-registers are always numbered above `num_regs`, which may be increased by the description, constant numbers can not be used for pseudos. They must be numbered relative to `num_regs` instead.
- The description will not describe pseudo-registers, so the architecture must call `set_tdesc_pseudo_register_name`, `set_tdesc_pseudo_register_type`, and `set_tdesc_pseudo_register_reggroup_p` to supply routines describing pseudo registers. These routines will be passed internal register numbers, so the same routines used for the gdbarch equivalents are usually suitable.

13 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.

13.1 Managing Execution State

A target vector can be completely inactive (not pushed on the target stack), active but not running (pushed, but not connected to a fully manifested inferior), or completely active (pushed, with an accessible inferior). Most targets are only completely inactive or completely active, but some support persistent connections to a target even when the target has exited or not yet started.

For example, connecting to the simulator using `target sim` does not create a running program. Neither registers nor memory are accessible until `run`. Similarly, after `kill`, the program can not continue executing. But in both cases GDB remains connected to the simulator, and target-specific commands are directed to the simulator.

A target which only supports complete activation should push itself onto the stack in its `to_open` routine (by calling `push_target`), and unpush itself from the stack in its `to_mourn_inferior` routine (by calling `unpush_target`).

A target which supports both partial and complete activation should still call `push_target` in `to_open`, but not call `unpush_target` in `to_mourn_inferior`. Instead, it should call either `target_mark_running` or `target_mark_exited` in its `to_open`, depending on whether the target is fully active after connection. It should also call `target_mark_running` any time the inferior becomes fully active (e.g. in `to_create_inferior` and `to_attach`), and `target_mark_exited` when the inferior becomes inactive (in `to_mourn_inferior`). The target should also make sure to call `target_mourn_inferior` from its `to_kill`, to return the target to inactive state.

13.2 Existing Targets

13.2.1 File Targets

Both executables and core files have target vectors.

13.2.2 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 '*cpu*`-stub.c`'. Many operating systems, embedded targets, emulators, and simulators already have a GDB stub built into them, and maintenance of the remote protocol must be careful to preserve compatibility.

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.

13.2.3 ROM Monitor Interface

13.2.4 Custom Protocols

13.2.5 Transport Layer

13.2.6 Builtin Simulator

14 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`', '`NAT_GENERATED_FILES`', 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.

14.1 ptrace

14.2 /proc

14.3 win32

14.4 shared libraries

14.5 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’.

I386_USE_GENERIC_WATCHPOINTS
: An x86-based machine can define this to use the generic x86 watchpoint support; see Chapter 3 [Algorithms], page 4.

`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.

15 Support Libraries

15.1 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.

15.2 opcodes

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

15.3 readline

The `readline` library provides a set of functions for use by applications that allow users to edit command lines as they are typed in.

15.4 libiberty

The `libiberty` library provides a set of functions and features that integrate and improve on functionality found in modern operating systems. Broadly speaking, such features can be divided into three groups: supplemental functions (functions that may be missing in some environments and operating systems), replacement functions (providing a uniform and easier to use interface for commonly used standard functions), and extensions (which provide additional functionality beyond standard functions).

GDB uses various features provided by the `libiberty` library, for instance the C++ demangler, the IEEE floating format support functions, the input options parser '`getopt`', the '`obstack`' extension, and other functions.

15.4.1 `obstacks` in GDB

The obstack mechanism provides a convenient way to allocate and free chunks of memory. Each obstack is a pool of memory that is managed like a stack. Objects (of any nature, size and alignment) are allocated and freed in a LIFO fashion on an obstack (see `libiberty`'s documentation for a more detailed explanation of `obstacks`).

The most noticeable use of the `obstacks` in GDB is in object files. There is an obstack associated with each internal representation of an object file. Lots of things get allocated on these `obstacks`: dictionary entries, blocks, blockvectors, symbols, minimal symbols, types, vectors of fundamental types, class fields of types, object files section lists, object files section offset lists, line tables, symbol tables, partial symbol tables, string tables, symbol table private data, macros tables, debug information sections and entries, import and export lists (som), unwind information (hppa), dwarf2 location expressions data. Plus various strings such as directory names strings, debug format strings, names of types.

An essential and convenient property of all data on `obstacks` is that memory for it gets allocated (with `obstack_alloc`) at various times during a debugging session, but it is released all at once using the `obstack_free` function. The `obstack_free` function takes a pointer to where in the stack it must start the deletion from (much like the cleanup chains have a pointer to where to start the cleanups). Because of the stack like structure of the `obstacks`, this allows to free only a top portion of the obstack. There are a few instances in GDB where such thing happens. Calls to `obstack_free` are done after some local data is allocated to the obstack. Only the local data is deleted from the obstack. Of course this assumes that nothing between the `obstack_alloc` and the `obstack_free` allocates anything else on the same obstack. For this reason it is best and safest to use temporary `obstacks`.

Releasing the whole obstack is also not safe per se. It is safe only under the condition that we know the `obstacks` memory is no longer needed. In GDB we get rid of the `obstacks` only when we get rid of the whole objfile(s), for instance upon reading a new symbol file.

15.5 gnu-regex

Regex conditionals.

`C_ALLOCA`

`NFAILURES`
`RE_NREGS`

`SIGN_EXTEND_CHAR`
`SWITCH_ENUM_BUG`
`SYNTAX_TABLE`
`Sword`

`sparc`

15.6 Array Containers

Often it is necessary to manipulate a dynamic array of a set of objects. C forces some bookkeeping on this, which can get cumbersome and repetitive. The '`vec.h`' file contains macros for defining and using a typesafe vector type. The functions defined will be inlined when compiling, and so the abstraction cost should be zero. Domain checks are added to detect programming errors.

An example use would be an array of symbols or section information. The array can be grown as symbols are read in (or preallocated), and the accessor macros provided keep care of all the necessary bookkeeping. Because the arrays are type safe, there is no danger of accidentally mixing up the contents. Think of these as C++ templates, but implemented in C.

Because of the different behavior of structure objects, scalar objects and of pointers, there are three flavors of vector, one for each of these variants. Both the structure object and pointer variants pass pointers to objects around — in the former case the pointers are stored into the vector and in the latter case the pointers are dereferenced and the objects copied into the vector. The scalar object variant is suitable for `int`-like objects, and the vector elements are returned by value.

There are both `index` and `iterate` accessors. The iterator returns a boolean iteration condition and updates the iteration variable passed by reference. Because the iterator will be inlined, the address-of can be optimized away.

The vectors are implemented using the trailing array idiom, thus they are not resizeable without changing the address of the vector object itself. This means you cannot have variables or fields of vector type — always use a pointer to a vector. The one exception is the final field of a structure, which could be a vector type. You will have to use the `embedded_size` & `embedded_init` calls to create such objects, and they will probably not be resizeable (so don't use the *safe* allocation variants). The trailing array idiom is used (rather than a pointer to an array of data), because, if we allow `NULL` to also represent an empty vector, empty vectors occupy minimal space in the structure containing them.

Each operation that increases the number of active elements is available in *quick* and *safe* variants. The former presumes that there is sufficient allocated space for the operation to

succeed (it dies if there is not). The latter will reallocate the vector, if needed. Reallocation causes an exponential increase in vector size. If you know you will be adding N elements, it would be more efficient to use the reserve operation before adding the elements with the *quick* operation. This will ensure there are at least as many elements as you ask for, it will exponentially increase if there are too few spare slots. If you want reserve a specific number of slots, but do not want the exponential increase (for instance, you know this is the last allocation), use a negative number for reservation. You can also create a vector of a specific size from the get go.

You should prefer the push and pop operations, as they append and remove from the end of the vector. If you need to remove several items in one go, use the truncate operation. The insert and remove operations allow you to change elements in the middle of the vector. There are two remove operations, one which preserves the element ordering `ordered_remove`, and one which does not `unordered_remove`. The latter function copies the end element into the removed slot, rather than invoke a memmove operation. The `lower_bound` function will determine where to place an item in the array using insert that will maintain sorted order.

If you need to directly manipulate a vector, then the `address` accessor will return the address of the start of the vector. Also the `space` predicate will tell you whether there is spare capacity in the vector. You will not normally need to use these two functions.

Vector types are defined using a `DEF_VEC_{O,P,I}(`*typename*`)` macro. Variables of vector type are declared using a `VEC(`*typename*`)` macro. The characters `O`, `P` and `I` indicate whether *typename* is an object (`O`), pointer (`P`) or integral (`I`) type. Be careful to pick the correct one, as you'll get an awkward and inefficient API if you use the wrong one. There is a check, which results in a compile-time warning, for the `P` and `I` versions, but there is no check for the `O` versions, as that is not possible in plain C.

An example of their use would be,

```
DEF_VEC_P(tree);   // non-managed tree vector.

struct my_struct {
  VEC(tree) *v;      // A (pointer to) a vector of tree pointers.
};

struct my_struct *s;

if (VEC_length(tree, s->v)) { we have some contents }
VEC_safe_push(tree, s->v, decl); // append some decl onto the end
for (ix = 0; VEC_iterate(tree, s->v, ix, elt); ix++)
  { do something with elt }
```

The '`vec.h`' file provides details on how to invoke the various accessors provided. They are enumerated here:

`VEC_length`
: Return the number of items in the array,

`VEC_empty`
: Return true if the array has no elements.

`VEC_last`
`VEC_index`
: Return the last or arbitrary item in the array.

`VEC_iterate`
: Access an array element and indicate whether the array has been traversed.

`VEC_alloc`
`VEC_free`
: Create and destroy an array.

`VEC_embedded_size`
`VEC_embedded_init`
: Helpers for embedding an array as the final element of another struct.

`VEC_copy`
: Duplicate an array.

`VEC_space`
: Return the amount of free space in an array.

`VEC_reserve`
: Ensure a certain amount of free space.

`VEC_quick_push`
`VEC_safe_push`
: Append to an array, either assuming the space is available, or making sure that it is.

`VEC_pop`
: Remove the last item from an array.

`VEC_truncate`
: Remove several items from the end of an array.

`VEC_safe_grow`
: Add several items to the end of an array.

`VEC_replace`
: Overwrite an item in the array.

`VEC_quick_insert`
`VEC_safe_insert`
: Insert an item into the middle of the array. Either the space must already exist, or the space is created.

`VEC_ordered_remove`
`VEC_unordered_remove`
: Remove an item from the array, preserving order or not.

`VEC_block_remove`
: Remove a set of items from the array.

`VEC_address`
: Provide the address of the first element.

`VEC_lower_bound`
: Binary search the array.

15.7 include

16 Coding

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

16.1 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).

16.2 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:

`struct gdbarch_data *` [Architecture Function]
gdbarch_data_register_pre_init (`gdbarch_data_pre_init_ftype *pre_init`)

pre_init is used to, on-demand, allocate an initial value for a per-architecture data-pointer using the architecture's obstack (passed in as a parameter). Since *pre_init* can be called during architecture creation, it is not parameterized with the architecture. and must not call modules that use per-architecture data.

`struct gdbarch_data *` [Architecture Function]
gdbarch_data_register_post_init (`gdbarch_data_post_init_ftype *post_init`)

post_init is used to obtain an initial value for a per-architecture data-pointer *after*. Since *post_init* is always called after architecture creation, it both receives the fully initialized architecture and is free to call modules that use per-architecture data (care needs to be taken to ensure that those other modules do not try to call back to this module as that will create in cycles in the initialization call graph).

These functions return a `struct gdbarch_data` that is used to identify the per-architecture data-pointer added for that module.

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

`void *` **gdbarch_data** (`struct gdbarch *`*gdbarch*`, struct gdbarch_data *`*data_handle*`)` [Architecture Function]

Given the architecture *arch* and module data handle *data_handle* (returned by `gdbarch_data_register_pre_init` or `gdbarch_data_register_post_init`), this function returns the current value of the per-architecture data-pointer. If the data pointer is `NULL`, it is first initialized by calling the corresponding *pre_init* or *post_init* method.

The examples below assume the following definitions:

```
struct nozel { int total; };
static struct gdbarch_data *nozel_handle;
```

A module can extend the architecture vector, adding additional per-architecture data, using the *pre_init* method. The module's per-architecture data is then initialized during architecture creation.

In the below, the module's per-architecture *nozel* is added. An architecture can specify its nozel by calling `set_gdbarch_nozel` from `gdbarch_init`.

```
static void *
nozel_pre_init (struct obstack *obstack)
{
  struct nozel *data = OBSTACK_ZALLOC (obstack, struct nozel);
  return data;
}
extern void
set_gdbarch_nozel (struct gdbarch *gdbarch, int total)
{
  struct nozel *data = gdbarch_data (gdbarch, nozel_handle);
  data->total = nozel;
}
```

A module can on-demand create architecture dependent data structures using `post_init`.

In the below, the nozel's total is computed on-demand by `nozel_post_init` using information obtained from the architecture.

```
static void *
nozel_post_init (struct gdbarch *gdbarch)
{
  struct nozel *data = GDBARCH_OBSTACK_ZALLOC (gdbarch, struct nozel);
  nozel->total = gdbarch... (gdbarch);
  return data;
}
extern int
nozel_total (struct gdbarch *gdbarch)
{
  struct nozel *data = gdbarch_data (gdbarch, nozel_handle);
  return data->total;
}
```

16.3 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.

16.4 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.

16.4.1 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.

16.4.2 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 `xstrprintf`.

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

16.4.3 Compiler Warnings

With few exceptions, developers should avoid the configuration option '`--disable-werror`' when building GDB. The exceptions are listed in the file '`gdb/MAINTAINERS`'. The default, when building with GCC, is '`--enable-werror`'.

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 are treated as errors.

The current list of warning flags includes:

'`-Wall`' Recommended GCC warnings.

'`-Wdeclaration-after-statement`'
GCC 3.x (and later) and C99 allow declarations mixed with code, but GCC 2.x and C89 do not.

'`-Wpointer-arith`'
'`-Wformat-nonliteral`'
Non-literal format strings, with a few exceptions, are bugs - they might contain unintended user-supplied format specifiers. 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`.

'`-Wno-pointer-sign`'
In version 4.0, GCC began warning about pointer argument passing or assignment even when the source and destination differed only in signedness. However, most GDB code doesn't distinguish carefully between `char` and `unsigned char`. In early 2006 the GDB developers decided correcting these warnings wasn't worth the time it would take.

'`-Wno-unused-parameter`'
Due to the way that GDB is implemented many functions have unused parameters. Consequently this warning is avoided. 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.

'`-Wno-unused`'
'`-Wno-switch`'
'`-Wno-char-subscripts`'
These are warnings which might be useful for GDB, but are currently too noisy to enable with '`-Werror`'.

16.4.4 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;
```

16.4.5 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.

16.4.6 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`'.

16.4.7 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.

16.4.8 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`*.*

16.4.9 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`'). These headers should be relatively independent: they should use only macros defined by '`configure`', the compiler, or the host; they should include only system headers; they should refer only to system types. They may be shared between multiple programs, e.g. GDB and GDBSERVER.

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

16.4.10 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
```

16.4.11 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 *value*s, 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 filesystems 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 `#ifdef`s everywhere, but by defining the `target_ops` structure and having a current target (as well as a stack of targets below it, for memory references). Whenever something needs to be done that depends on which remote interface we are using, a flag in the current target_ops structure is tested (e.g., `target_has_stack`), or a function is called through a pointer in the current target_ops structure. In this way, when a new remote interface is added, only one module needs to be touched—the one that actually implements the new remote interface. Other examples of attribute-structs are BFD access to multiple kinds of object file formats, or GDB's access to multiple source languages.

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

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`.

17 Porting GDB

Most of the work in making GDB compile on a new machine is in specifying the configuration of the machine. Porting a new architecture to GDB can be broken into a number of steps.

- Ensure a BFD exists for executables of the target architecture in the '`bfd`' directory. If one does not exist, create one by modifying an existing similar one.
- Implement a disassembler for the target architecture in the '`opcodes`' directory.
- Define the target architecture in the '`gdb`' directory (see Section 11.11 [Adding a New Target], page 69). Add the pattern for the new target to '`configure.tgt`' with the names of the files that contain the code. By convention the target architecture definition for an architecture *arch* is placed in '*`arch`*`-tdep.c`'.

  Within '*`arch`*`-tdep.c`' define the function `_initialize_`*`arch`*`_tdep` which calls `gdbarch_register` to create the new `struct gdbarch` for the architecture.
- If a new remote target is needed, consider adding a new remote target by defining a function `_initialize_remote_`*`arch`*. However if at all possible use the GDB *Remote Serial Protocol* for this and implement the server side protocol independently with the target.
- If desired implement a simulator in the '`sim`' directory. This should create the library '`libsim.a`' implementing the interface in '`remote-sim.h`' (found in the '`include`' directory).
- Build and test. If desired, lobby the GDB steering group to have the new port included in the main distribution!
- Add a description of the new architecture to the main GDB user guide (see section "Configuration Specific Information" in *Debugging with* GDB).

18 Versions and Branches

18.1 Versions

GDB's version is determined by the file '`gdb/version.in`' and takes one of the following forms:

major.minor
major.minor.patchlevel
: an official release (e.g., 6.2 or 6.2.1)

major.minor.patchlevel.YYYYMMDD
: a snapshot taken at *YYYY-MM-DD*-gmt (e.g., 6.1.50.20020302, 6.1.90.20020304, or 6.1.0.20020308)

major.minor.patchlevel.YYYYMMDD-cvs
: a CVS check out drawn on *YYYY-MM-DD* (e.g., 6.1.50.20020302-cvs, 6.1.90.20020304-cvs, or 6.1.0.20020308-cvs)

major.minor.patchlevel.YYYYMMDD (*vendor*)
: a vendor specific release of GDB, that while based on *major.minor.patchlevel.YYYYMMDD*, may include additional changes

GDB's mainline uses the *major* and *minor* version numbers from the most recent release branch, with a *patchlevel* of 50. At the time each new release branch is created, the mainline's *major* and *minor* version numbers are updated.

GDB's release branch is similar. When the branch is cut, the *patchlevel* is changed from 50 to 90. As draft releases are drawn from the branch, the *patchlevel* is incremented. Once the first release (*major.minor*) has been made, the *patchlevel* is set to 0 and updates have an incremented *patchlevel*.

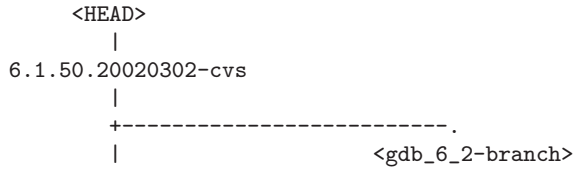
For snapshots, and CVS check outs, it is also possible to identify the CVS origin:

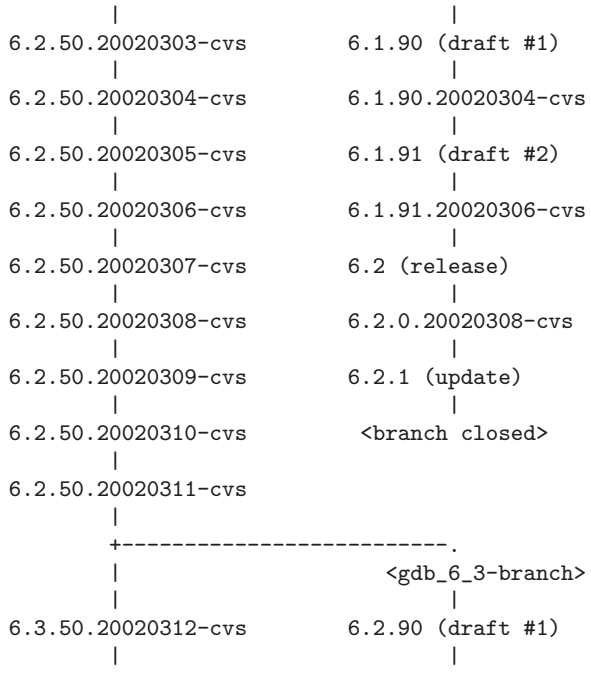
major.minor.50.*YYYYMMDD*
: drawn from the HEAD of mainline CVS (e.g., 6.1.50.20020302)

major.minor.90.*YYYYMMDD*
major.minor.91.*YYYYMMDD* ...
: drawn from a release branch prior to the release (e.g., 6.1.90.20020304)

major.minor.0.*YYYYMMDD*
major.minor.1.*YYYYMMDD* ...
: drawn from a release branch after the release (e.g., 6.2.0.20020308)

If the previous GDB version is 6.1 and the current version is 6.2, then, substituting 6 for *major* and 1 or 2 for *minor*, here's an illustration of a typical sequence:





18.2 Release Branches

GDB draws a release series (6.2, 6.2.1, ...) from a single release branch, and identifies that branch using the CVS branch tags:

`gdb_`*major*`_`*minor*`-YYYYMMDD-branchpoint`
`gdb_`*major*`_`*minor*`-branch`
`gdb_`*major*`_`*minor*`-YYYYMMDD-release`

Pragmatics: To help identify the date at which a branch or release is made, both the branchpoint and release tags include the date that they are cut (YYYYMMDD) in the tag. The branch tag, denoting the head of the branch, does not need this.

18.3 Vendor Branches

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 identifier). E.g., '`6.2widgit2`', or '`6.2 (Widgit Inc Patch 2)`'.

18.4 Experimental Branches

18.4.1 Guidelines

GDB permits the creation of branches, cut from the CVS repository, for experimental development. Branches make it possible for developers to share preliminary work, and maintainers to examine significant new developments.

The following are a set of guidelines for creating such branches:

a branch has an owner
: The owner can set further policy for a branch, but may not change the ground rules. In particular, they can set a policy for commits (be it adding more reviewers or deciding who can commit).

all commits are posted
: All changes committed to a branch shall also be posted to the GDB patches mailing list. While commentary on such changes are encouraged, people should remember that the changes only apply to a branch.

all commits are covered by an assignment
: This ensures that all changes belong to the Free Software Foundation, and avoids the possibility that the branch may become contaminated.

a branch is focused
: A focused branch has a single objective or goal, and does not contain unnecessary or irrelevant changes. Cleanups, where identified, being be pushed into the mainline as soon as possible.

a branch tracks mainline
: This keeps the level of divergence under control. It also keeps the pressure on developers to push cleanups and other stuff into the mainline.

a branch shall contain the entire GDB *module*
: The GDB module `gdb` should be specified when creating a branch (branches of individual files should be avoided). See [Tags], page 92.

a branch shall be branded using '`version.in`'
: The file '`gdb/version.in`' shall be modified so that it identifies the branch *owner* and branch *name*, e.g., '`6.2.50.20030303_owner_name`' or '`6.2 (Owner Name)`'.

18.4.2 Tags

To simplify the identification of GDB branches, the following branch tagging convention is strongly recommended:

owner_name`-`*YYYYMMDD*`-branchpoint`
owner_name`-`*YYYYMMDD*`-branch`
: The branch point and corresponding branch tag. *YYYYMMDD* is the date that the branch was created. A branch is created using the sequence:

```
cvs rtag owner_name-YYYYMMDD-branchpoint gdb
cvs rtag -b -r owner_name-YYYYMMDD-branchpoint \
   owner_name-YYYYMMDD-branch gdb
```

owner_name`-`*yyyymmdd*`-mergepoint`
: The tagged point, on the mainline, that was used when merging the branch on *yyyymmdd*. To merge in all changes since the branch was cut, use a command sequence like:

```
cvs rtag owner_name-yyyymmdd-mergepoint gdb
cvs update \
```

```
-jowner_name-YYYYMMDD-branchpoint
-jowner_name-yyyymmdd-mergepoint
```

Similar sequences can be used to just merge in changes since the last merge.

For further information on CVS, see Concurrent Versions System.

19 Start of New Year Procedure

At the start of each new year, the following actions should be performed:

- Rotate the ChangeLog file

  The current '`ChangeLog`' file should be renamed into '`ChangeLog-YYYY`' where YYYY is the year that has just passed. A new '`ChangeLog`' file should be created, and its contents should contain a reference to the previous ChangeLog. The following should also be preserved at the end of the new ChangeLog, in order to provide the appropriate settings when editing this file with Emacs:

  ```
  Local Variables:
  mode: change-log
  left-margin: 8
  fill-column: 74
  version-control: never
  coding: utf-8
  End:
  ```

- Add an entry for the newly created ChangeLog file ('`ChangeLog-YYYY`') in '`gdb/config/djgpp/fnchange.lst`'.
- Update the copyright year in the startup message

  Update the copyright year in:

  - file '`top.c`', function `print_gdb_version`
  - file '`gdbserver/server.c`', function `gdbserver_version`
  - file '`gdbserver/gdbreplay.c`', function `gdbreplay_version`
- Run the '`copyright.sh`' script to add the new year in the copyright notices of most source files. This script requires Emacs 22 or later to be installed.
- The new year also needs to be added manually in all other files that are not already taken care of by the '`copyright.sh`' script:
  - '`*.s`'
  - '`*.f`'
  - '`*.f90`'
  - '`*.igen`'
  - '`*.ac`'
  - '`*.texi`'
  - '`*.texinfo`'
  - '`*.tex`'
  - '`*.defs`'
  - '`*.1`'

20 Releasing GDB

20.1 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.

20.2 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.*

20.3 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.

20.3.1 Review the bug data base

Close anything obviously fixed.

20.3.2 Check all cross targets build

The targets are listed in 'gdb/MAINTAINERS'.

20.4 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:sourceware.org:/cvs/src rtag \
-D $D-gmt gdb_$V-$D-branchpoint insight
cvs -f -d :ext:sourceware.org:/cvs/src rtag
-D 2002-03-03-gmt gdb_5_2-2002-03-03-branchpoint insight
$  ^echo ^^
...
$  echo cvs -f -d :ext:sourceware.org:/cvs/src rtag \
-b -r gdb_$V-$D-branchpoint gdb_$V-branch insight
cvs -f -d :ext:sourceware.org:/cvs/src rtag \
-b -r gdb_5_2-2002-03-03-branchpoint gdb_5_2-branch insight
$  ^echo ^^
...
$
```

- By using *-D YYYY-MM-DD-gmt*, the branch is forced to an exact date/time.
- The trunk is first tagged so that the branch point can easily be found.
- Insight, which includes GDB, is 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:sourceware.org:/cvs/src co \
-r gdb_$V-branch src/gdb/version.in
cvs -f -d :ext:sourceware.org:/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 Discussion mailing list and GDB Testers 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.

20.5 Stabilize the branch

Something goes here.

20.6 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.

20.6.1 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@sourceware.org.

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 documented in the toplevel '`README-maintainer-mode`' file. It is very unlikely that the version of `autoconf` installed in system directories (e.g., '`/usr/bin/autoconf`') is correct.

Check out the relevant modules:

```
$ for m in gdb insight
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
```

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 src-release $m.tar )
done
```

If the top level source directory does not have ‘src-release’ (GDB version 5.3.1 or earlier), try these commands instead:

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

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 suppressed). Fixing it would be nice though.*

Create compressed versions of the release

```
$  cp */src/*.tar .
$  cp */src/*.bz2 .
$  ls -F
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.

20.6.2 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)
```

20.6.3 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://sourceware.org/pub/gdb/snapshots/branch`
4. Notify the relevant mailing lists (`gdb@sourceware.org` and `gdb-testers@sourceware.org` that the candidate is available.

20.6.4 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

20.6.5 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.

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`.

20.7 Post release

Remove any `OBSOLETE` code.

21 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. The tests themselves are calls to various `Tcl` procs; the framework runs all the procs and summarizes the passes and fails.

21.1 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
```

To run a specific test script, type:

```
make check RUNTESTFLAGS='tests'
```

where *tests* is a list of test script file names, separated by spaces.

If you use GNU make, you can use its '`-j`' option to run the testsuite in parallel. This can greatly reduce the amount of time it takes for the testsuite to run. In this case, if you set `RUNTESTFLAGS` then, by default, the tests will be run serially even under '`-j`'. You can override this and force a parallel run by setting the `make` variable `FORCE_PARALLEL` to any non-empty value. Note that the parallel *`make check`* assumes that you want to run the entire testsuite, so it is not compatible with some dejagnu options, like '`--directory`'.

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.

DejaGNU supports separate build, host, and target machines. However, some GDB test scripts do not work if the build machine and the host machine are not the same. In such an environment, these scripts will give a result of "UNRESOLVED", like this:

```
UNRESOLVED: gdb.base/example.exp: This test script does not work on a remote host.
```

Sometimes it is convenient to get a transcript of the commands which the testsuite sends to GDB. For example, if GDB crashes during testing, a transcript can be used to more easily reconstruct the failure when running GDB under GDB.

You can instruct the GDB testsuite to write transcripts by setting the DejaGNU variable `TRANSCRIPT` (to any value) before invoking `runtest` or *`make check`*. The transcripts will be written into DejaGNU's output directory. One transcript will be made for each invocation of GDB; they will be named '`transcript.`*n*', where *n* is an integer. The first line of the transcript file will show how GDB was invoked; each subsequent line is a command sent as input to GDB.

```
make check RUNTESTFLAGS=TRANSCRIPT=y
```

Note that the transcript is not always complete. In particular, tests of completion can yield partial command lines.

21.2 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++ (`#ifdef`s are allowed if necessary, for instance for prototypes).

'`gdb.lang`'
: Language-specific tests for any language *lang* besides C. Examples are '`gdb.cp`' 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.

21.3 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. Even if GDB has several valid responses to a command, you can use `gdb_test_multiple`. Like `gdb_test`, `gdb_test_multiple` recognizes internal errors and unexpected prompts.

Do not write tests which expect a literal tab character from GDB. On some operating systems (e.g. OpenBSD) the TTY layer expands tabs to spaces, so by the time GDB's output reaches expect the tab is gone.

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.

22 Hints

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

22.1 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.

22.2 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.

22.3 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.

22.4 Build Script

The script '`gdb_buildall.sh`' builds GDB with flag '`--enable-targets=all`' set. This builds GDB with all supported targets activated. This helps testing GDB when doing changes that affect more than one architecture and is much faster than using '`gdb_mbuild.sh`'.

After building GDB the script checks which architectures are supported and then switches the current architecture to each of those to get information about the architecture. The test results are stored in log files in the directory the script was called from.

Appendix A GDB Currently available observers

A.1 Implementation rationale

An *observer* is an entity which is interested in being notified when GDB reaches certain states, or certain events occur in GDB. The entity being observed is called the *subject*. To receive notifications, the observer attaches a callback to the subject. One subject can have several observers.

'`observer.c`' implements an internal generic low-level event notification mechanism. This generic event notification mechanism is then re-used to implement the exported high-level notification management routines for all possible notifications.

The current implementation of the generic observer provides support for contextual data. This contextual data is given to the subject when attaching the callback. In return, the subject will provide this contextual data back to the observer as a parameter of the callback.

Note that the current support for the contextual data is only partial, as it lacks a mechanism that would deallocate this data when the callback is detached. This is not a problem so far, as this contextual data is only used internally to hold a function pointer. Later on, if a certain observer needs to provide support for user-level contextual data, then the generic notification mechanism will need to be enhanced to allow the observer to provide a routine to deallocate the data when attaching the callback.

The observer implementation is also currently not reentrant. In particular, it is therefore not possible to call the attach or detach routines during a notification.

A.2 Debugging

Observer notifications can be traced using the command '`set debug observer 1`' (see section "Optional messages about internal happenings" in *Debugging with GDBN*).

A.3 `normal_stop` Notifications

GDB notifies all `normal_stop` observers when the inferior execution has just stopped, the associated messages and annotations have been printed, and the control is about to be returned to the user.

Note that the `normal_stop` notification is not emitted when the execution stops due to a breakpoint, and this breakpoint has a condition that is not met. If the breakpoint has any associated commands list, the commands are executed after the notification is emitted.

The following interfaces are available to manage observers:

`extern struct observer *observer_attach_`*`event`* `(observer_`*`event`*`_ftype *`*`f`*`)` [Function]
Using the function *f*, create an observer that is notified when ever *event* occurs, return the observer.

`extern void observer_detach_`*`event`* `(struct observer *`*`observer`*`);` [Function]
Remove *observer* from the list of observers to be notified when *event* occurs.

`extern void observer_notify_`*`event`* `(void);` [Function]
Send a notification to all *event* observers.

The following observable events are defined:

`void` **normal_stop** `(struct bpstats *`*`bs`*`, int `*`print_frame`*`)` [Function]
The inferior has stopped for real. The *bs* argument describes the breakpoints were are stopped at, if any. Second argument *print_frame* non-zero means display the location where the inferior has stopped.

`void` **target_changed** (`struct target_ops *target`) [Function]
The target's register contents have changed.

`void` **executable_changed** (`void`) [Function]
The executable being debugged by GDB has changed: The user decided to debug a different program, or the program he was debugging has been modified since being loaded by the debugger (by being recompiled, for instance).

`void` **inferior_created** (`struct target_ops *objfile, int from_tty`) [Function]
GDB has just connected to an inferior. For '`run`', GDB calls this observer while the inferior is still stopped at the entry-point instruction. For '`attach`' and '`core`', GDB calls this observer immediately after connecting to the inferior, and before any information on the inferior has been printed.

`void` **solib_loaded** (`struct so_list *solib`) [Function]
The shared library specified by *solib* has been loaded. Note that when GDB calls this observer, the library's symbols probably haven't been loaded yet.

`void` **solib_unloaded** (`struct so_list *solib`) [Function]
The shared library specified by *solib* has been unloaded. Note that when GDB calls this observer, the library's symbols have not been unloaded yet, and thus are still available.

`void` **new_objfile** (`struct objfile *objfile`) [Function]
The symbol file specified by *objfile* has been loaded. Called with *objfile* equal to `NULL` to indicate previously loaded symbol table data has now been invalidated.

`void` **new_thread** (`struct thread_info *t`) [Function]
The thread specified by *t* has been created.

`void` **thread_exit** (`struct thread_info *t, int silent`) [Function]
The thread specified by *t* has exited. The *silent* argument indicates that GDB is removing the thread from its tables without wanting to notify the user about it.

`void` **thread_stop_requested** (`ptid_t ptid`) [Function]
An explicit stop request was issued to *ptid*. If *ptid* equals *minus_one_ptid*, the request applied to all threads. If `ptid_is_pid(ptid)` returns true, the request applied to all threads of the process pointed at by *ptid*. Otherwise, the request applied to the single thread pointed at by *ptid*.

`void` **target_resumed** (`ptid_t ptid`) [Function]
The target was resumed. The *ptid* parameter specifies which thread was resume, and may be RESUME_ALL if all threads are resumed.

`void` **about_to_proceed** (`void`) [Function]
The target is about to be proceeded.

`void` **breakpoint_created** (`int bpnum`) [Function]
A new breakpoint has been created. The argument *bpnum* is the number of the newly-created breakpoint.

`void` **breakpoint_deleted** (`int` *bpnum*) [Function]
A breakpoint has been destroyed. The argument *bpnum* is the number of the newly-destroyed breakpoint.

`void` **breakpoint_modified** (`int` *bpnum*) [Function]
A breakpoint has been modified in some way. The argument *bpnum* is the number of the modified breakpoint.

`void` **tracepoint_created** (`int` *tpnum*) [Function]
A new tracepoint has been created. The argument *tpnum* is the number of the newly-created tracepoint.

`void` **tracepoint_deleted** (`int` *tpnum*) [Function]
A tracepoint has been destroyed. The argument *tpnum* is the number of the newly-destroyed tracepoint.

`void` **tracepoint_modified** (`int` *tpnum*) [Function]
A tracepoint has been modified in some way. The argument *tpnum* is the number of the modified tracepoint.

`void` **architecture_changed** (`struct gdbarch *`*newarch*) [Function]
The current architecture has changed. The argument *newarch* is a pointer to the new architecture.

`void` **thread_ptid_changed** (`ptid_t` *old_ptid*, `ptid_t` *new_ptid*) [Function]
The thread's ptid has changed. The *old_ptid* parameter specifies the old value, and *new_ptid* specifies the new value.

`void` **inferior_appeared** (`int` *pid*) [Function]
GDB has attached to a new inferior identified by *pid*.

`void` **inferior_exit** (`int` *pid*) [Function]
Either GDB detached from the inferior, or the inferior exited. The argument *pid* identifies the inferior.

`void` **memory_changed** (`CORE_ADDR` *addr*, `int` *len*, `const bfd_byte` `*`*data*) [Function]
Bytes from *data* to *data* `+` *len* have been written to the current inferior at *addr*.

`void` **test_notification** (`int` *somearg*) [Function]
This observer is used for internal testing. Do not use. See testsuite/gdb.gdb/observer.exp.

Appendix B GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

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

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

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to 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, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

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. (Thus, 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. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

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

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

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

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

3. COPYING IN QUANTITY

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

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

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

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

4. MODIFICATIONS

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

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

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

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

D. Preserve all the copyright notices of the Document.

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

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

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

H. Include an unaltered copy of this License.

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

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

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

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

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

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

O. Preserve any Warranty Disclaimers.

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

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

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

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

5. COMBINING DOCUMENTS

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

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

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

6. COLLECTIONS OF DOCUMENTS

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

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

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, is called

an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket 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, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

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.

B.1 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 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

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

Index

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X