Working in GDB

A guide to the internals of the GNU debugger

John Gilmore
Cygnus Support

Cygnus Support
Revision: 1.41
TEXinfo 2023-09-19.19

1 The `README` File

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

2 Defining a New Host or Target Architecture

When building support for a new host and/or target, much of the work you need to do is handled by specifying configuration files; see Chapter 3 [Adding a New Configuration], page 2. Further work can be divided into "host-dependent" (see Chapter 4 [Adding a New Host], page 3) and "target-dependent" (see Chapter 6 [Adding a New Target], page 6). The following discussion is meant to explain the difference between hosts and targets.

What is considered "host-dependent" versus "target-dependent"?

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

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

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

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

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

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

The dependent information in GDB is organized into files by naming conventions.

Host-Dependent Files

`config/*.mh`
: Sets Makefile parameters

`xm-*.h`
: Global #include's and #define's and definitions

`*-xdep.c`
: Global variables and functions

Native-Dependent Files

`config/*.mh`
: Sets Makefile parameters (for *both* host and native)

`nm-*.h` #include's and #define's and definitions. This file is only included by the small number of modules that need it, so beware of doing feature-test #define's from its macros.

`*-nat.c` global variables and functions

Target-Dependent Files

`config/*.mt`
Sets Makefile parameters

`tm-*.h` Global #include's and #define's and definitions

`*-tdep.c` Global variables and functions

At this writing, most supported hosts have had their host and native dependencies sorted out properly. There are a few stragglers, which can be recognized by the absence of NATDEPFILES lines in their `config/*.mh`.

3 Adding a New Configuration

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

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

```
./config.sub xxx
```

and

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

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

Now, go to the `bfd` directory and create a new file `bfd/hosts/h-xxx.h`. Examine the other `h-*.h` files as templates, and create one that brings in the right include files for your system, and defines any host-specific macros needed by BFD, the Binutils, GNU LD, or the Opcodes directories. (They all share the bfd `hosts` directory and the `configure.host` file.)

Then edit `bfd/configure.host`. Add a line to recognize your *`xarch-xvend-xos`* configuration, and set `my_host` to *xxx* when you recognize it. This will cause your file `h-xxx.h` to be linked to `sysdep.h` at configuration time. When creating the line that recognizes your configuration, only match the fields that you really need to match; e.g. don't match match the architecture or manufacturer if the OS is sufficient to distinguish the configuration that your `h-xxx.h` file supports. Don't match the manufacturer name unless you really need to. This should make future ports easier.

Also, if this host requires any changes to the Makefile, create a file `bfd/config/xxx.mh`, which includes the required lines.

It's possible that the `libiberty` and `readline` directories won't need any changes for your configuration, but if they do, you can change the `configure.in` file there to recognize your system and map to an `mh-xxx` file. Then add `mh-xxx` to the `config/` subdirectory, to set any makefile variables you need. The only current options in there are things like '`-DSYSV`'. (This `mh-xxx` naming convention differs from elsewhere in GDB, by historical accident. It should be cleaned up so that all such files are called `xxx.mh`.)

Aha! Now to configure GDB itself! Edit `gdb/configure.in` to recognize your system and set `gdb_host` to *xxx*, and (unless your desired target is already available) also set `gdb_target` to something appropriate (for instance, *xxx*). To handle new hosts, modify the segment after the comment '`# per-host`'; to handle new targets, modify after '`# per-target`'.

Finally, you'll need to specify and define GDB's host-, native-, and target-dependent `.h` and `.c` files used for your configuration; the next two chapters discuss those.

4 Adding a New Host

Once you have specified a new configuration for your host (see Chapter 3 [Adding a New Configuration], page 2), there are three remaining pieces to making GDB work on a new machine. First, you have to make it host on the new machine (compile there, handle that machine's terminals properly, etc). If you will be cross-debugging to some other kind of system that's already supported, you are done.

If you want to use GDB to debug programs that run on the new machine, you have to get it to understand the machine's object files, symbol files, and interfaces to processes; see Chapter 6 [Adding a New Target], page 6, and see Chapter 5 [Adding a New Native Configuration], page 4,

Several files control GDB's configuration for host systems:

`gdb/config/mh-xxx`
: Specifies Makefile fragments needed when hosting on machine *xxx*. In particular, this lists the required machine-dependent object files, by defining '`XDEPFILES=...`'. Also specifies the header file which describes host *xxx*, by defining '`XM_FILE= xm-xxx.h`'. You can also define '`CC`', '`REGEX`' and '`REGEX1`', '`SYSV_DEFINE`', '`XM_CFLAGS`', '`XM_ADD_FILES`', '`XM_CLIBS`', '`XM_CDEPS`', etc.; see `Makefile.in`.

`gdb/xm-xxx.h`
: (`xm.h` is a link to this file, created by configure). Contains C macro definitions describing the host system environment, such as byte order, host C compiler and library, ptrace support, and core file structure. Crib from existing `xm-*.h` files to create a new one.

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

Generic Host Support Files

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

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

`ser-bsd.c`
: This contains serial line support for Berkeley-derived Unix systems.

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

`ser-termios.c`
: This contains serial line support for System V-derived Unix systems.

Now, you are now ready to try configuring GDB to compile using your system as its host. From the top level (above `bfd`, `gdb`, etc), do:

```
./configure xxx +target=vxworks960
```

This will configure your system to cross-compile for VxWorks on the Intel 960, which is probably not what you really want, but it's a test case that works at this stage. (You haven't set up to be able to debug programs that run *on* *xxx* yet.)

If this succeeds, you can try building it all with:

```
make
```

Repeat until the program configures, compiles, links, and runs. When run, it won't be able to do much (unless you have a VxWorks/960 board on your network) but you will know that the host support is pretty well done.

Good luck! Comments and suggestions about this section are particularly welcome; send them to '`bug-gdb@prep.ai.mit.edu`'.

5 Adding a New Native Configuration

If you are making GDB run native on the *xxx* machine, you have plenty more work to do. Several files control GDB's configuration for native support:

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

`gdb/nm-xxx.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. Crib from existing `nm-*.h` files to create a new one. Code that needs these definitions will have to `#include "nm.h"` explicitly, since it is not included by `defs.h`.

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

Generic Native Support Files

There are some "generic" versions of routines that can be used by various systems. These can be customized in various ways by macros defined in your `nm-xxx.h` file. If these routines work for the *xxx* 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 `xxx-nat.c`, and put `xxx-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.

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

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

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

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

Then back in GDB, you need a matching routine called `fetch_core_registers()`. If you can use the generic one, it's in `coredep.c`; if not, it's in your *xxx*`-nat.c` file. It will be passed a char pointer to the entire "registers" segment, its length, and a zero; or a char pointer to the entire "regs2" segment, its length, and a 2. The routine should suck out the supplied register values and install them into GDB's "registers" array. (See Chapter 2 [Defining a New Host or Target Architecture], page 1, for more info about this.)

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

6 Adding a New Target

For a new target called *ttt*, first specify the configuration as described in Chapter 3 [Adding a New Configuration], page 2. If your new target is the same as your new host, you've probably already done that.

A variety of files specify attributes of the GDB target environment:

`gdb/config/ttt.mt`
: Contains a Makefile fragment specific to this target. Specifies what object files are needed for target *ttt*, by defining '`TDEPFILES=...`'. Also specifies the header file which describes *ttt*, by defining '`TM_FILE= tm-ttt.h`'. You can also define '`TM_CFLAGS`', '`TM_CLIBS`', '`TM_CDEPS`', and other Makefile variables here; see `Makefile.in`.

`gdb/tm-ttt.h`
: (`tm.h` is a link to this file, created by configure). Contains macro definitions about the target machine's registers, stack frame format and instructions. Crib from existing `tm-*.h` files when building a new one.

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

`gdb/exec.c`
: Defines functions for accessing files that are executable on the target system. These functions open and examine an exec file, extract data from one, write data to one, print information about one, etc. Now that executable files are handled with BFD, every target should be able to use the generic exec.c rather than its own custom code.

`gdb/`*`arch`*`-pinsn.c`
: Prints (disassembles) the target machine's instructions. This file is usually shared with other target machines which use the same processor, which is why it is *`arch`*`-pinsn.c` rather than *`ttt`*`-pinsn.c`.

`gdb/`*`arch`*`-opcode.h`
: Contains some large initialized data structures describing the target machine's instructions. This is a bit strange for a `.h` file, but it's OK since it is only included in one place. *`arch`*`-opcode.h` is shared between the debugger and the assembler, if the GNU assembler has been ported to the target machine.

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

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

When adding support for a new target machine, there are various areas of support that might need change, or might be OK.

If you are using an existing object file format (a.out or COFF), there is probably little to be done. See `bfd/doc/bfd.texinfo` for more information on writing new a.out or COFF versions.

If you need to add a new object file format, you are beyond the scope of this document right now. Look at the structure of the a.out and COFF support, build a transfer vector (`xvec`) for your new format, and start populating it with routines. Add it to the list in `bfd/targets.c`.

If you are adding a new operating system for an existing CPU chip, add a `tm-`*`xos`*`.h` file that describes the operating system facilities that are unusual (extra symbol table info; the breakpoint instruction needed; etc). Then write a `tm-`*`xarch-xos`*`.h` that just `#include`s `tm-`*`xarch`*`.h` and `tm-`*`xos`*`.h`. (Now that we have three-part configuration names, this will probably get revised to separate the *xos* configuration from the *xarch* configuration.)

7 Adding a Source Language to GDB

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

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

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

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

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

Add any evaluation routines, if necessary

If you need new opcodes (that represent the operations of the language), add them to the enumerated type in `expression.h`. Add support code for these operations in `eval.c:evaluate_subexp()`. Add cases for new opcodes in two functions from `parse.c`: `prefixify_subexp()` and `length_of_subexp()`. These compute the number of `exp_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 `expprint.c:print_subexp()` to handle any new expression opcodes you have added to `expression.h`. Also, add the printed representations of your operators to `op_print_tab`.

Add a place of call

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

Use macros to trim code

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

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

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

Edit `Makefile.in`

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

8 Configuring GDB for Release

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

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

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

This procedure requires:

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

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

TEMPORARY RELEASE PROCEDURE FOR DOCUMENTATION

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

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

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

9 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. (See Chapter 11 [Symbol Reading], page 11.)

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.

10 Binary File Descriptor Library Support for GDB

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.

11 Symbol Reading

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

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

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

The functions supplied by each module are:

xxx`_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 *xxx*`_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 *xxx*`_symfile_init`, but it can call `error` if it detects an unavoidable problem.

xxx`_new_init()`

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

xxx`_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 *xxx*`_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 *xxx*_symfile_read is called, these psymtabs will contain a pointer to a function *xxx*`_psymtab_to_symtab`, which can be called from any point in the GDB symbol-handling code.

xxx`_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.

12 Cleanups

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

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

Syntax:

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

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

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

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

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

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

13 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.

14 Frames

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

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

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

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

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

15 Coding Style

GDB is generally written using the GNU coding standards, as described in `standards.texi`, which you can get from the Free Software Foundation. There are some additional considerations for GDB maintainers that reflect the unique environment and style of GDB maintenance. If you follow these guidelines, GDB will be more consistent and easier to maintain.

GDB's policy on the use of prototypes is that prototypes are used to *declare* functions but never to *define* them. Simple macros are used in the declarations, so that a non-ANSI compiler can compile GDB without trouble. The simple macro calls are used like this:

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

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

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

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

All external functions should have a `PARAMS` declaration in a header file that callers include. All static functions should have such a declaration near the top of their source file.

We don't have a gcc option that will properly check that these rules have been followed, but it's GDB policy, and we periodically check it using the tools available (plus manual labor), and clean up any remnants.

16 Host Conditionals

When GDB is configured and compiled, various macros are defined or left undefined, to control compilation based on the attributes of the host system. These macros and their meanings are:

NOTE: For now, both host and target conditionals are here. Eliminate target conditionals from this list as they are identified.

`ALIGN_SIZE`
: alloca.c

`BLOCK_ADDRESS_FUNCTION_RELATIVE`
: dbxread.c

`GDBINIT_FILENAME`
: main.c

`KERNELDEBUG`
: tm-hppa.h

`MEM_FNS_DECLARED`
: defs.h

`NO_SYS_FILE`
: dbxread.c

`PYRAMID_CONTROL_FRAME_DEBUGGING`
: pyr-xdep.c

`SIGWINCH_HANDLER_BODY`
: utils.c

`1`
: buildsym.c

`1`
: dbxread.c

`1`
: dbxread.c

`1`
: buildsym.c

`1`
: dwarfread.c

`1`
: valops.c

`1`
: valops.c

`1`
: pyr-xdep.c

`ADDITIONAL_OPTIONS`
: main.c

`ADDITIONAL_OPTION_CASES`
: main.c

`ADDITIONAL_OPTION_HANDLER`
: main.c

`ADDITIONAL_OPTION_HELP`
: main.c

`ADDR_BITS_REMOVE`
: defs.h

`AIX_BUGGY_PTRACE_CONTINUE`
: infptrace.c

`ALIGN_STACK_ON_STARTUP`
: main.c

`ALTOS` altos-xdep.c

`ALTOS_AS` xm-altos.h

`ASCII_COFF`
: remote-adapt.c

`BADMAG` coffread.c

`BCS` tm-delta88.h

`BEFORE_MAIN_LOOP_HOOK`
: main.c

`BELIEVE_PCC_PROMOTION`
: coffread.c

`BELIEVE_PCC_PROMOTION_TYPE`
: stabsread.c

`BIG_ENDIAN`
: defs.h

`BITS_BIG_ENDIAN`
: defs.h

`BKPT_AT_MAIN`
: solib.c

`BLOCK_ADDRESS_ABSOLUTE`
: dbxread.c

`BPT_VECTOR`
: tm-68k.h

`BREAKPOINT`
: tm-68k.h

`BREAKPOINT_DEBUG`
breakpoint.c

`BROKEN_LARGE_ALLOCA`
Avoid large `alloca`'s. For example, on sun's, Large alloca's fail because the attempt to increase the stack limit in main() fails because shared libraries are allocated just below the initial stack limit. The SunOS kernel will not allow the stack to grow into the area occupied by the shared libraries.

`BSTRING` regex.c

`CALL_DUMMY`
valops.c

`CALL_DUMMY_LOCATION`
inferior.h

`CALL_DUMMY_STACK_ADJUST`
valops.c

`CANNOT_FETCH_REGISTER`
hppabsd-xdep.c

`CANNOT_STORE_REGISTER`
findvar.c

`CFRONT_PRODUCER`
dwarfread.c

`CHILD_PREPARE_TO_STORE`
inftarg.c

`CLEAR_DEFERRED_STORES`
inflow.c

`CLEAR_SOLIB`
objfiles.c

`COFF_ENCAPSULATE`
hppabsd-tdep.c

`COFF_FORMAT`
symm-tdep.c

`COFF_NO_LONG_FILE_NAMES`
coffread.c

`CORE_NEEDS_RELOCATION`
stack.c

`CPLUS_MARKER`
cplus-dem.c

`CREATE_INFERIOR_HOOK`
infrun.c

`C_ALLOCA` regex.c

`C_GLBLREG`
: coffread.c

`DAMON`
: xcoffexec.c

`DBXREAD_ONLY`
: partial-stab.h

`DBX_PARM_SYMBOL_CLASS`
: stabsread.c

`DEBUG`
: remote-adapt.c

`DEBUG_INFO`
: partial-stab.h

`DEBUG_PTRACE`
: hppabsd-xdep.c

`DECR_PC_AFTER_BREAK`
: breakpoint.c

`DEFAULT_PROMPT`
: main.c

`DELTA88`
: m88k-xdep.c

`DEV_TTY`
: symmisc.c

`DGUX`
: m88k-xdep.c

`DISABLE_UNSETTABLE_BREAK`
: breakpoint.c

`DONT_USE_REMOTE`
: remote.c

`DO_DEFERRED_STORES`
: infrun.c

`DO_REGISTERS_INFO`
: infcmd.c

`END_OF_TEXT_DEFAULT`
: dbxread.c

`EXTERN`
: buildsym.h

`EXTRACT_RETURN_VALUE`
: tm-68k.h

`EXTRACT_STRUCT_VALUE_ADDRESS`
: values.c

`EXTRA_FRAME_INFO`
: frame.h

`EXTRA_SYMTAB_INFO`
: symtab.h

`FILES_INFO_HOOK`
target.c

`FIXME` coffread.c

`FLOAT_INFO`
infcmd.c

`FOPEN_RB` defs.h

`FPO_REGNUM`
a68v-xdep.c

`FPC_REGNUM`
mach386-xdep.c

`FP_REGNUM`
parse.c

`FRAMELESS_FUNCTION_INVOCATION`
blockframe.c

`FRAME_ARGS_ADDRESS_CORRECT`
stack.c

`FRAME_CHAIN_COMBINE`
blockframe.c

`FRAME_CHAIN_VALID`
frame.h

`FRAME_CHAIN_VALID_ALTERNATE`
frame.h

`FRAME_FIND_SAVED_REGS`
stack.c

`FRAME_GET_BASEREG_VALUE`
frame.h

`FRAME_NUM_ARGS`
tm-68k.h

`FRAME_SPECIFICATION_DYADIC`
stack.c

`FUNCTION_EPILOGUE_SIZE`
coffread.c

`F_OK` xm-ultra3.h

`GCC2_COMPILED_FLAG_SYMBOL`
dbxread.c

`GCC_COMPILED_FLAG_SYMBOL`
dbxread.c

`GCC_MANGLE_BUG`
symtab.c

`GCC_PRODUCER`
: dwarfread.c

`GET_SAVED_REGISTER`
: findvar.c

`GPLUS_PRODUCER`
: dwarfread.c

`GR64_REGNUM`
: remote-adapt.c

`GR64_REGNUM`
: remote-mm.c

`HANDLE_RBRAC`
: partial-stab.h

`HAVE_68881`
: m68k-tdep.c

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

`HAVE_REGISTER_WINDOWS`
: findvar.c

`HAVE_SIGSETMASK`
: main.c

`HAVE_TERMIO`
: inflow.c

`HEADER_SEEK_FD`
: arm-tdep.c

`HOSTING_ONLY`
: xm-rtbsd.h

`HOST_BYTE_ORDER`
: ieee-float.c

`HPUX_ASM` xm-hp300hpux.h

`HPUX_VERSION_5`
: hp300ux-xdep.c

`HP_OS_BUG`
: infrun.c

`I80960` remote-vx.c

`IBM6000_HOST`
: breakpoint.c

`IBM6000_TARGET`
: buildsym.c

`IEEE_DEBUG`
: ieee-float.c

`IEEE_FLOAT`
: valprint.c

`IGNORE_SYMBOL`
: dbxread.c

`INIT_EXTRA_FRAME_INFO`
: blockframe.c

`INIT_EXTRA_SYMTAB_INFO`
: symfile.c

`INIT_FRAME_PC`
: blockframe.c

`INNER_THAN`
: valops.c

`INT_MAX` defs.h

`INT_MIN` defs.h

`IN_GDB` i960-pinsn.c

`IN_SIGTRAMP`
: infrun.c

`IN_SOLIB_TRAMPOLINE`
: infrun.c

`ISATTY` main.c

`IS_TRAPPED_INTERNALVAR`
: values.c

`KERNELDEBUG`
: dbxread.c

`KERNEL_DEBUGGING`
: tm-ultra3.h

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

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

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

`LCC_PRODUCER`
: dwarfread.c

`LITTLE_ENDIAN`
: defs.h

`LOG_FILE` remote-adapt.c

`LONGERNAMES`
: cplus-dem.c

`LONGEST` defs.h

`LONG_LONG`
: defs.h

`LONG_MAX` defs.h

`LSEEK_NOT_LINEAR`
: source.c

`L_LNNO32` coffread.c

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

`MACHKERNELDEBUG`
: hppabsd-tdep.c

`MAIN` cplus-dem.c

`MAINTENANCE`
: dwarfread.c

`MAINTENANCE_CMDS`
: breakpoint.c

`MAINTENANCE_CMDS`
: maint.c

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

`MIPSEL` mips-tdep.c

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

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

`MONO` ser-go32.c

`MOTOROLA` xm-altos.h

`NAMES_HAVE_UNDERSCORE`
: coffread.c

`NBPG` altos-xdep.c

`NEED_POSIX_SETPGID`
infrun.c

`NEED_TEXT_START_END`
exec.c

`NFAILURES`
regex.c

`NNPC_REGNUM`
infrun.c

`NORETURN` defs.h

`NOTDEF` regex.c

`NOTDEF` remote-adapt.c

`NOTDEF` remote-mm.c

`NOTICE_SIGNAL_HANDLING_CHANGE`
infrun.c

`NO_DEFINE_SYMBOL`
xcoffread.c

`NO_HIF_SUPPORT`
remote-mm.c

`NO_JOB_CONTROL`
signals.h

`NO_MALLOC_CHECK`
utils.c

`NO_MMALLOC`
utils.c

`NO_MMALLOC`
objfiles.c

`NO_MMALLOC`
utils.c

`NO_SIGINTERRUPT`
remote-adapt.c

`NO_SINGLE_STEP`
infptrace.c

`NO_TYPEDEFS`
xcoffread.c

`NO_TYPEDEFS`
xcoffread.c

`NPC_REGNUM`
infcmd.c

`NS32K_SVC_IMMED_OPERANDS`
ns32k-opcode.h

`NUMERIC_REG_NAMES`
mips-tdep.c

`N_SETV` dbxread.c

`N_SET_MAGIC`
hppabsd-tdep.c

`NaN` tm-umax.h

`ONE_PROCESS_WRITETEXT`
breakpoint.c

`O_BINARY` exec.c

`O_RDONLY` xm-ultra3.h

`PC` convx-opcode.h

`PCC_SOL_BROKEN`
dbxread.c

`PC_IN_CALL_DUMMY`
inferior.h

`PC_LOAD_SEGMENT`
stack.c

`PC_REGNUM`
parse.c

`PRINT_RANDOM_SIGNAL`
infcmd.c

`PRINT_REGISTER_HOOK`
infcmd.c

`PRINT_TYPELESS_INTEGER`
valprint.c

`PROCESS_LINENUMBER_HOOK`
buildsym.c

`PROLOGUE_FIRSTLINE_OVERLAP`
infrun.c

`PSIGNAL_IN_SIGNAL_H`
defs.h

`PS_REGNUM`
parse.c

`PTRACE_ARG3_TYPE`
: inferior.h

`PTRACE_FP_BUG`
: mach386-xdep.c

`PT_ATTACH`
: hppabsd-xdep.c

`PT_DETACH`
: hppabsd-xdep.c

`PT_KILL`
: infptrace.c

`PUSH_ARGUMENTS`
: valops.c

`PYRAMID_CONTROL_FRAME_DEBUGGING`
: pyr-xdep.c

`PYRAMID_CORE`
: pyr-xdep.c

`PYRAMID_PTRACE`
: pyr-xdep.c

`REGISTER_BYTES`
: remote.c

`REGISTER_NAMES`
: tm-29k.h

`REG_STACK_SEGMENT`
: exec.c

`REG_STRUCT_HAS_ADDR`
: findvar.c

`RE_NREGS`
: regex.h

`R_FP`
: dwarfread.c

`R_OK`
: xm-altos.h

`SDB_REG_TO_REGNUM`
: coffread.c

`SEEK_END`
: state.c

`SEEK_SET`
: state.c

`SEM`
: coffread.c

`SET_STACK_LIMIT_HUGE`
: When defined, stack limits will be raised to their maximum. Use this if your host supports `setrlimit` and you have trouble with `stringtab` in `dbxread.c`.

  Also used in `fork-child.c` to return stack limits before child processes are forked.

`SHELL_COMMAND_CONCAT`
infrun.c

`SHELL_FILE`
infrun.c

`SHIFT_INST_REGS`
breakpoint.c

`SIGN_EXTEND_CHAR`
regex.c

`SIGTRAP_STOP_AFTER_LOAD`
infrun.c

`SKIP_PROLOGUE`
tm-68k.h

`SKIP_PROLOGUE_FRAMELESS_P`
blockframe.c

`SKIP_TRAMPOLINE_CODE`
infrun.c

`SOLIB_ADD`
core.c

`SOLIB_CREATE_INFERIOR_HOOK`
infrun.c

`SOME_NAMES_HAVE_DOT`
minsyms.c

`SP_REGNUM`
parse.c

`STAB_REG_TO_REGNUM`
stabsread.h

`STACK_ALIGN`
valops.c

`STACK_DIRECTION`
alloca.c

`START_INFERIOR_TRAPS_EXPECTED`
infrun.c

`STOP_SIGNAL`
main.c

`STORE_RETURN_VALUE`
tm-68k.h

`SUN4_COMPILER_FEATURE`
infrun.c

`SUN_FIXED_LBRAC_BUG`
: dbxread.c

`SVR4_SHARED_LIBS`
: solib.c

`SWITCH_ENUM_BUG`
: regex.c

`SYM1`
: tm-ultra3.h

`SYMBOL_RELOADING_DEFAULT`
: symfile.c

`SYNTAX_TABLE`
: regex.c

`Sword`
: regex.c

`TDESC`
: infrun.c

`TIOCGETC`
: inflow.c

`TIOCGLTC`
: inflow.c

`TIOCGPGRP`
: inflow.c

`TIOCLGET`
: inflow.c

`TIOCLSET`
: inflow.c

`TIOCNOTTY`
: inflow.c

`TM_FILE_OVERRIDE`
: defs.h

`T_ARG`
: coffread.c

`T_VOID`
: coffread.c

`UINT_MAX`
: defs.h

`UPAGES`
: altos-xdep.c

`USER`
: m88k-tdep.c

`USE_GAS`
: xm-news.h

`USE_O_NOCTTY`
: inflow.c

`USE_STRUCT_CONVENTION`
: values.c

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

`USIZE` xm-m88k.h

`U_FPSTATE`
i386-xdep.c

`VARIABLES_INSIDE_BLOCK`
dbxread.c

`WRS_ORIG` remote-vx.c

`_LANG_c` language.c

`_LANG_m2` language.c

`__GNUC__` news-xdep.c

`__GO32__` inflow.c

`__HAVE_68881__`
m68k-stub.c

`__HPUX_ASM__`
xm-hp300hpux.h

`__INT_VARARGS_H`
printcmd.c

`__not_on_pyr_yet`
pyr-xdep.c

`alloca` defs.h

`const` defs.h

`GOULD_PN` gould-pinsn.c

`emacs` alloca.c

`hp800` xm-hppabsd.h

`hpux` hppabsd-core.c

`lint` valarith.c

`longest_to_int`
defs.h

`mc68020` m68k-stub.c

`notdef` gould-pinsn.c

`ns32k_opcodeT`
ns32k-opcode.h

`sgi` mips-tdep.c

`sparc` regex.c

`static` alloca.c

`sun` m68k-tdep.c

`sun386` tm-sun386.h

`test` regex.c

`ultrix` xm-mips.h

`volatile` defs.h

`x_name` coffread.c

`x_zeroes` coffread.c

17 Target Conditionals

When GDB is configured and compiled, various macros are defined or left undefined, to control compilation based on the attributes of the target system. These macros and their meanings are:

NOTE: For now, both host and target conditionals are here. Eliminate host conditionals from this list as they are identified.

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

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

`ALIGN_SIZE`
: alloca.c

`BLOCK_ADDRESS_FUNCTION_RELATIVE`
: dbxread.c

`GDBINIT_FILENAME`
: main.c

`KERNELDEBUG`
: tm-hppa.h

`MEM_FNS_DECLARED`
: defs.h

`NO_SYS_FILE`
: dbxread.c

`PYRAMID_CONTROL_FRAME_DEBUGGING`
: pyr-xdep.c

`SIGWINCH_HANDLER_BODY`
: utils.c

`ADDITIONAL_OPTIONS`
: main.c

`ADDITIONAL_OPTION_CASES`
: main.c

`ADDITIONAL_OPTION_HANDLER`
main.c

`ADDITIONAL_OPTION_HELP`
main.c

`ADDR_BITS_REMOVE`
defs.h

`ALIGN_STACK_ON_STARTUP`
main.c

`ALTOS` altos-xdep.c

`ALTOS_AS` xm-altos.h

`ASCII_COFF`
remote-adapt.c

`BADMAG` coffread.c

`BCS` tm-delta88.h

`BEFORE_MAIN_LOOP_HOOK`
main.c

`BELIEVE_PCC_PROMOTION`
coffread.c

`BELIEVE_PCC_PROMOTION_TYPE`
stabsread.c

`BIG_ENDIAN`
defs.h

`BITS_BIG_ENDIAN`
defs.h

`BKPT_AT_MAIN`
solib.c

`BLOCK_ADDRESS_ABSOLUTE`
dbxread.c

`BPT_VECTOR`
tm-68k.h

`BREAKPOINT`
tm-68k.h

`BREAKPOINT_DEBUG`
breakpoint.c

`BSTRING` regex.c

`CALL_DUMMY`
valops.c

`CALL_DUMMY_LOCATION`
: inferior.h

`CALL_DUMMY_STACK_ADJUST`
: valops.c

`CANNOT_FETCH_REGISTER`
: hppabsd-xdep.c

`CANNOT_STORE_REGISTER`
: findvar.c

`CFRONT_PRODUCER`
: dwarfread.c

`CHILD_PREPARE_TO_STORE`
: inftarg.c

`CLEAR_DEFERRED_STORES`
: inflow.c

`CLEAR_SOLIB`
: objfiles.c

`COFF_ENCAPSULATE`
: hppabsd-tdep.c

`COFF_FORMAT`
: symm-tdep.c

`COFF_NO_LONG_FILE_NAMES`
: coffread.c

`CORE_NEEDS_RELOCATION`
: stack.c

`CPLUS_MARKER`
: cplus-dem.c

`CREATE_INFERIOR_HOOK`
: infrun.c

`C_ALLOCA` regex.c

`C_GLBLREG`
: coffread.c

`DAMON` xcoffexec.c

`DBXREAD_ONLY`
: partial-stab.h

`DBX_PARM_SYMBOL_CLASS`
: stabsread.c

`DEBUG` remote-adapt.c

`DEBUG_INFO`
: partial-stab.h

`DEBUG_PTRACE`
: hppabsd-xdep.c

`DECR_PC_AFTER_BREAK`
: breakpoint.c

`DEFAULT_PROMPT`
: main.c

`DELTA88` m88k-xdep.c

`DEV_TTY` symmisc.c

`DGUX` m88k-xdep.c

`DISABLE_UNSETTABLE_BREAK`
: breakpoint.c

`DONT_USE_REMOTE`
: remote.c

`DO_DEFERRED_STORES`
: infrun.c

`DO_REGISTERS_INFO`
: infcmd.c

`END_OF_TEXT_DEFAULT`
: dbxread.c

`EXTERN` buildsym.h

`EXTRACT_RETURN_VALUE`
: tm-68k.h

`EXTRACT_STRUCT_VALUE_ADDRESS`
: values.c

`EXTRA_FRAME_INFO`
: frame.h

`EXTRA_SYMTAB_INFO`
: symtab.h

`FILES_INFO_HOOK`
: target.c

`FIXME` coffread.c

`FLOAT_INFO`
: infcmd.c

`FOPEN_RB` defs.h

`FPO_REGNUM`
: a68v-xdep.c

`FPC_REGNUM`
: mach386-xdep.c

`FP_REGNUM`
parse.c

`FPU` Unused? 6-oct-92 rich@cygnus.com. FIXME.

`FRAMELESS_FUNCTION_INVOCATION`
blockframe.c

`FRAME_ARGS_ADDRESS_CORRECT`
stack.c

`FRAME_CHAIN_COMBINE`
blockframe.c

`FRAME_CHAIN_VALID`
frame.h

`FRAME_CHAIN_VALID_ALTERNATE`
frame.h

`FRAME_FIND_SAVED_REGS`
stack.c

`FRAME_GET_BASEREG_VALUE`
frame.h

`FRAME_NUM_ARGS`
tm-68k.h

`FRAME_SPECIFICATION_DYADIC`
stack.c

`FUNCTION_EPILOGUE_SIZE`
coffread.c

`F_OK` xm-ultra3.h

`GCC2_COMPILED_FLAG_SYMBOL`
dbxread.c

`GCC_COMPILED_FLAG_SYMBOL`
dbxread.c

`GCC_MANGLE_BUG`
symtab.c

`GCC_PRODUCER`
dwarfread.c

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

`GDB_TARGET_IS_MACH386`
mach386-xdep.c

`GDB_TARGET_IS_SUN3`
: a68v-xdep.c

`GDB_TARGET_IS_SUN386`
: sun386-xdep.c

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

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

`GET_SAVED_REGISTER`
: findvar.c

`GPLUS_PRODUCER`
: dwarfread.c

`GR64_REGNUM`
: remote-adapt.c

`GR64_REGNUM`
: remote-mm.c

`HANDLE_RBRAC`
: partial-stab.h

`HAVE_68881`
: m68k-tdep.c

`HAVE_REGISTER_WINDOWS`
: findvar.c

`HAVE_SIGSETMASK`
: main.c

`HAVE_TERMIO`
: inflow.c

`HEADER_SEEK_FD`
: arm-tdep.c

`HOSTING_ONLY`
: xm-rtbsd.h

`HOST_BYTE_ORDER`
: ieee-float.c

`HPUX_ASM` xm-hp300hpux.h

`HPUX_VERSION_5`
: hp300ux-xdep.c

`HP_OS_BUG`
: infrun.c

`I80960`
: remote-vx.c

`IBM6000_HOST`
: breakpoint.c

`IBM6000_TARGET`
: buildsym.c

`IEEE_DEBUG`
: ieee-float.c

`IEEE_FLOAT`
: valprint.c

`IGNORE_SYMBOL`
: dbxread.c

`INIT_EXTRA_FRAME_INFO`
: blockframe.c

`INIT_EXTRA_SYMTAB_INFO`
: symfile.c

`INIT_FRAME_PC`
: blockframe.c

`INNER_THAN`
: valops.c

`INT_MAX`
: defs.h

`INT_MIN`
: defs.h

`IN_GDB`
: i960-pinsn.c

`IN_SIGTRAMP`
: infrun.c

`IN_SOLIB_TRAMPOLINE`
: infrun.c

`ISATTY`
: main.c

`IS_TRAPPED_INTERNALVAR`
: values.c

`KERNELDEBUG`
: dbxread.c

`KERNEL_DEBUGGING`
: tm-ultra3.h

`LCC_PRODUCER`
: dwarfread.c

`LITTLE_ENDIAN`
defs.h

`LOG_FILE` remote-adapt.c

`LONGERNAMES`
cplus-dem.c

`LONGEST` defs.h

`LONG_LONG`
defs.h

`LONG_MAX` defs.h

`L_LNNO32` coffread.c

`MACHKERNELDEBUG`
hppabsd-tdep.c

`MAIN` cplus-dem.c

`MAINTENANCE`
dwarfread.c

`MAINTENANCE_CMDS`
breakpoint.c

`MAINTENANCE_CMDS`
maint.c

`MIPSEL` mips-tdep.c

`MOTOROLA` xm-altos.h

`NAMES_HAVE_UNDERSCORE`
coffread.c

`NBPG` altos-xdep.c

`NEED_POSIX_SETPGID`
infrun.c

`NEED_TEXT_START_END`
exec.c

`NFAILURES`
regex.c

`NNPC_REGNUM`
infrun.c

`NORETURN` defs.h

`NOTDEF` regex.c

`NOTDEF` remote-adapt.c

`NOTDEF` remote-mm.c

`NOTICE_SIGNAL_HANDLING_CHANGE`
: infrun.c

`NO_DEFINE_SYMBOL`
: xcoffread.c

`NO_HIF_SUPPORT`
: remote-mm.c

`NO_JOB_CONTROL`
: signals.h

`NO_MALLOC_CHECK`
: utils.c

`NO_MMALLOC`
: utils.c

`NO_MMALLOC`
: objfiles.c

`NO_MMALLOC`
: utils.c

`NO_SIGINTERRUPT`
: remote-adapt.c

`NO_SINGLE_STEP`
: infptrace.c

`NO_TYPEDEFS`
: xcoffread.c

`NO_TYPEDEFS`
: xcoffread.c

`NPC_REGNUM`
: infcmd.c

`NS32K_SVC_IMMED_OPERANDS`
: ns32k-opcode.h

`NUMERIC_REG_NAMES`
: mips-tdep.c

`N_SETV` dbxread.c

`N_SET_MAGIC`
: hppabsd-tdep.c

`NaN` tm-umax.h

`ONE_PROCESS_WRITETEXT`
: breakpoint.c

`PC` convx-opcode.h

`PCC_SOL_BROKEN`
: dbxread.c

`PC_IN_CALL_DUMMY`
: inferior.h

`PC_LOAD_SEGMENT`
: stack.c

`PC_REGNUM`
: parse.c

`PRINT_RANDOM_SIGNAL`
: infcmd.c

`PRINT_REGISTER_HOOK`
: infcmd.c

`PRINT_TYPELESS_INTEGER`
: valprint.c

`PROCESS_LINENUMBER_HOOK`
: buildsym.c

`PROLOGUE_FIRSTLINE_OVERLAP`
: infrun.c

`PSIGNAL_IN_SIGNAL_H`
: defs.h

`PS_REGNUM`
: parse.c

`PTRACE_ARG3_TYPE`
: inferior.h

`PTRACE_FP_BUG`
: mach386-xdep.c

`PUSH_ARGUMENTS`
: valops.c

`REGISTER_BYTES`
: remote.c

`REGISTER_NAMES`
: tm-29k.h

`REG_STACK_SEGMENT`
: exec.c

`REG_STRUCT_HAS_ADDR`
: findvar.c

`RE_NREGS` regex.h

`R_FP` dwarfread.c

`R_OK` xm-altos.h

`SDB_REG_TO_REGNUM`
: coffread.c

`SEEK_END` state.c

`SEEK_SET` state.c

`SEM` coffread.c

`SET_STACK_LIMIT_HUGE`
infrun.c

`SHELL_COMMAND_CONCAT`
infrun.c

`SHELL_FILE`
infrun.c

`SHIFT_INST_REGS`
breakpoint.c

`SIGN_EXTEND_CHAR`
regex.c

`SIGTRAP_STOP_AFTER_LOAD`
infrun.c

`SKIP_PROLOGUE`
tm-68k.h

`SKIP_PROLOGUE_FRAMELESS_P`
blockframe.c

`SKIP_TRAMPOLINE_CODE`
infrun.c

`SOLIB_ADD`
core.c

`SOLIB_CREATE_INFERIOR_HOOK`
infrun.c

`SOME_NAMES_HAVE_DOT`
minsyms.c

`SP_REGNUM`
parse.c

`STAB_REG_TO_REGNUM`
stabsread.h

`STACK_ALIGN`
valops.c

`STACK_DIRECTION`
alloca.c

`START_INFERIOR_TRAPS_EXPECTED`
infrun.c

`STOP_SIGNAL`
main.c

`STORE_RETURN_VALUE`
tm-68k.h

`SUN4_COMPILER_FEATURE`
infrun.c

`SUN_FIXED_LBRAC_BUG`
dbxread.c

`SVR4_SHARED_LIBS`
solib.c

`SWITCH_ENUM_BUG`
regex.c

`SYM1` tm-ultra3.h

`SYMBOL_RELOADING_DEFAULT`
symfile.c

`SYNTAX_TABLE`
regex.c

`Sword` regex.c

`TARGET_BYTE_ORDER`
defs.h

`TARGET_CHAR_BIT`
defs.h

`TARGET_COMPLEX_BIT`
defs.h

`TARGET_DOUBLE_BIT`
defs.h

`TARGET_DOUBLE_COMPLEX_BIT`
defs.h

`TARGET_FLOAT_BIT`
defs.h

`TARGET_INT_BIT`
defs.h

`TARGET_LONG_BIT`
defs.h

`TARGET_LONG_DOUBLE_BIT`
defs.h

`TARGET_LONG_LONG_BIT`
defs.h

`TARGET_PTR_BIT`
defs.h

`TARGET_SHORT_BIT`
defs.h

`TDESC` infrun.c

`TM_FILE_OVERRIDE`
defs.h

`T_ARG` coffread.c

`T_VOID` coffread.c

`UINT_MAX` defs.h

`USER` m88k-tdep.c

`USE_GAS` xm-news.h

`USE_STRUCT_CONVENTION`
values.c

`USIZE` xm-m88k.h

`U_FPSTATE`
i386-xdep.c

`VARIABLES_INSIDE_BLOCK`
dbxread.c

`WRS_ORIG` remote-vx.c

`_LANG_c` language.c

`_LANG_m2` language.c

`__GO32__` inflow.c

`__HAVE_68881__`
m68k-stub.c

`__HPUX_ASM__`
xm-hp300hpux.h

`__INT_VARARGS_H`
printcmd.c

`__not_on_pyr_yet`
pyr-xdep.c

`GOULD_PN` gould-pinsn.c

`emacs` alloca.c

`hp800` xm-hppabsd.h

`hpux` hppabsd-core.c

`longest_to_int`
defs.h

`mc68020` m68k-stub.c

`ns32k_opcodeT`
: ns32k-opcode.h

`sgi`
: mips-tdep.c

`sparc`
: regex.c

`static`
: alloca.c

`sun`
: m68k-tdep.c

`sun386`
: tm-sun386.h

`test`
: regex.c

`x_name`
: coffread.c

`x_zeroes`
: coffread.c

18 Native Conditionals

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

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

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

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

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

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

`REGISTER_U_ADDR`
: Defines the offset of the registers in the "u area"; see Chapter 4 [Host], page 3.

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

`U_REGS_OFFSET`

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

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

19 Obsolete Conditionals

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

`STACK_END_ADDR`

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

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

Table of Contents