Working in GDB

A guide to the internals of the GNU debugger

John Gilmore
Cygnus Support

Cygnus Support
Revision: 1.84
TEXinfo 2.185

Scope of this Document

This document documents the internals of the GNU debugger, GDB. It is intended to document aspects of GDB which apply across many different parts of GDB (for example, see Chapter 20 [Coding Style], page 18), or which are global aspects of design (for example, what are the major modules and which files document them in detail?). Information which pertains to specific data structures, functions, variables, etc., should be put in comments in the source code, not here. It is more likely to get noticed and kept up to date there. Some of the information in this document should probably be moved into comments.

1 The 'README' File

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

2 Getting Started Working on GDB

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 (see Chapter 22 [Submitting Patches], page 20). 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 has a list of macros (see Chapter 23 [Host Conditionals], page 22, see Chapter 24 [Target Conditionals], page 29, see Chapter 25 [Native Conditionals], page 39, and see Chapter 26 [Obsolete Conditionals], page 40) with some documentation.
- Start with the header files. Once you some idea of how GDB's internal symbol tables are stored (see '`symtab.h`', '`gdbtypes.h`'), you will find it much easier to understand the code which uses and creates those symbol tables.
- You may wish to process the information you are getting somehow, to enhance your understanding of it. Summarize it, translate it to another language, add some (perhaps trivial or non-useful) feature to GDB, use the code to predict what a test case would do and write the test case and verify your prediction, etc. If you are reading code and your eyes are starting to glaze over, this is a sign you need to use a more active approach.

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

  Of course, this same technique can be used if you are just reading the code, rather than actually stepping through it. The same general principle applies—when the code you are looking at calls something else, just try to understand generally what the code being called does, rather than worrying about all its details.
- A good place to start when tracking down some particular area is with a command which invokes that feature. Suppose you want to know how single-stepping works. As a GDB user, you know that the `step` command invokes single-stepping. The command is invoked via command tables (see '`command.h`'); by convention the function which actually performs the command is formed by taking the name of the command and adding '`_command`', or in the case of an `info` subcommand, '`_info`'. For example, the `step` command invokes the `step_command` function and the `info display` command invokes `display_info`. When this convention is not followed, you might have to use `grep` or *`M-x tags-search`* in emacs, or run GDB on itself and set a breakpoint in `execute_command`.
- If all of the above fail, it may be appropriate to ask for information on `bug-gdb`. But *never* post a generic question like "I was wondering if anyone could give me some tips about understanding GDB"—if we had some magic secret we would put it in this manual. Suggestions for improving the manual are always welcome, of course.

Good luck!

3 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.

4 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 5 [Adding a New Configuration], page 4. Further work can be divided into "host-dependent" (see Chapter 6 [Adding a New Host], page 5) and "target-dependent" (see Chapter 8 [Adding a New Target], page 8). 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

'`config/*/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)

'config/*/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

'config/*/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'.

5 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 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.

6 Adding a New Host

Once you have specified a new configuration for your host (see Chapter 5 [Adding a New Configuration], page 4), 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 8 [Adding a New Target], page 8 and see Chapter 7 [Adding a New Native Configuration], page 6

Several files control GDB's configuration for host systems:

'`gdb/config/`*arch*`/`*xxx*`.mh`'
: 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/config/`*arch*`/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`'.

7 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/`*xarch*`/`*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 '`NAT_FILE= nm-`*xxx*`.h`'. You can also define '`NAT_CFLAGS`', '`NAT_ADD_FILES`', '`NAT_CLIBS`', '`NAT_CDEPS`', etc.; see '`Makefile.in`'.

'gdb/config/*arch*/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.

'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.

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

'core-aout.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, 'core-aout.c' will define the `register_addr()` function and use the macro in it. If you do not define `REGISTER_U_ADDR`, but you are using the standard `fetch_core_registers()`, you will need to define your own version of `register_addr()`, put it into your *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 '`core-aout.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 4 [Defining a New Host or Target Architecture], page 3, 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.

8 Adding a New Target

For a new target called *ttt*, first specify the configuration as described in Chapter 5 [Adding a New Configuration], page 4. 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/`*arch*`/`*ttt*`.mt`'
: Contains a Makefile fragment specific to this target. Specifies what object files are needed for target *ttt*, by defining '`TDEPFILES=`...'. Also specifies the header file which describes *ttt*, by defining '`TM_FILE= tm-`*ttt*`.h`'. You can also define '`TM_CFLAGS`', '`TM_CLIBS`', '`TM_CDEPS`', and other Makefile variables here; see '`Makefile.in`'.

'`gdb/config/`*arch*`/tm-`*ttt*`.h`'
: ('`tm.h`' is a link to this file, created by configure). Contains macro definitions about the target machine's registers, stack frame format and instructions. 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/config/`*arch*`/tm-`*arch*`.h`'
: This often exists to describe the basic layout of the target machine's processor chip (registers, stack, etc). If used, it is included by '`tm-`*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 must first add it to BFD. This is beyond the scope of this document right now. Basically you must build a transfer vector (of type `bfd_target`), which will mean writing all the required routines, and add it to the list in '`bfd/targets.c`'.

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/libxxx.h`', which is included by GDB.

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 `#includes` '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.)

9 Adding a Source Language to GDB

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

Create the expression parser.

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

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

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

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

Add any evaluation routines, if necessary

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

Update some existing code

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

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

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

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

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

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

Add a place of call

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

Use macros to trim code

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

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

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

Edit 'Makefile.in'

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

10 Configuring GDB for Release

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

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

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

This procedure requires:

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

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

TEMPORARY RELEASE PROCEDURE FOR DOCUMENTATION

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

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

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

11 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 14 [Symbol Reading], page 14.)

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.

12 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.

13 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.

14 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.

15 Cleanups

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

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

Syntax:

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

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

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

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

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

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

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

16 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.

17 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.

18 Remote Stubs

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

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

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

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

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

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

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

19 Longjmp Support

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

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

20 Coding Style

GDB is generally written using the GNU coding standards, as described in '`standards.texi`', which is available for anonymous FTP from GNU archive sites. 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.

21 Clean Design

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

- You can't assume the byte order of anything that comes from a target (including *values*, object files, and instructions). Such things must be byte-swapped using `SWAP_TARGET_AND_HOST` in GDB, or one of the swap routines defined in '`bfd.h`', such as `bfd_get_32`.
- You can't assume that you know what interface is being used to talk to the target system. All references to the target must go through the current `target_ops` vector.
- You can't assume that the host and target machines are the same machine (except in the "native" support modules). In particular, you can't assume that the target machine's header files will be available on the host machine. Target code must bring along its own header files – written from scratch or explicitly donated by their owner, to avoid copyright problems.
- Insertion of new `#ifdef`'s will be frowned upon. It's much better to write the code portably than to conditionalize it for various systems.
- New `#ifdef`'s which test for specific compilers or manufacturers or operating systems are unacceptable. All `#ifdef`'s should test for features. The information about which configurations contain which features should be segregated into the configuration files. Experience has proven far too often that a feature unique to one particular system often creeps into other systems; and that a conditional based on some predefined macro for your current system will become worthless over time, as new versions of your system come out that behave differently with regard to this feature.
- Adding code that handles specific architectures, operating systems, target interfaces, or hosts, is not acceptable in generic code. If a hook is needed at that point, invent a generic hook and define it for your configuration, with something like:

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

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

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

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

```
#endif
```

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

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

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

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

- *Do* write code that doesn't 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. In short, follow good programming practices for writing portable C code.

22 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 two main problems with getting your patches in are,

- The GDB maintainers will only install "cleanly designed" patches. You may not always agree on what is clean design. see Chapter 20 [Coding Style], page 18, see Chapter 21 [Clean Design], page 19.
- 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.

I don't know how to get past these problems except by continuing to try.

There are two issues here – technical and legal.

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 document for doing this by sending mail to `gnu@prep.ai.mit.edu` and asking for it. I recommend that people write in "All programs owned by the Free Software Foundation" as "NAME OF PROGRAM", so that changes in many programs (not just GDB, but GAS, Emacs, GCC, etc) can be contributed with only one piece of legalese pushed through the bureacracy and filed with the FSF. I can't start merging changes until this paperwork is received by the FSF (their rules, which I follow since I 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.texi or gdb/doc/gdbint.texi). If there are a lot of changes for a single feature, they can be split down into multiple messages.

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

The reason to send each change in a separate message is that I will not install some of the changes. They'll be returned to you with questions or comments. If I'm doing my job, my 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 other changes (which I *am* willing to accept) can be installed while one or more changes are being reworked. If multiple features are sent in a single message, I 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 I get a lot of bug reports and a lot of patches, and most of them don't get installed because I 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 if and when I scan back over the queue – which can literally take months sometimes. It's in both our interests to make patch installation easy – you get your changes installed, and I make some forward progress on GDB in a normal 12-hour day (instead of them having to wait until I have a 14-hour or 16-hour day to spend cleaning up patches before I can install them).

Please send patches to bug-gdb@prep.ai.mit.edu, if they are less than about 25,000 characters. If longer than that, either make them available somehow (e.g. anonymous FTP), and announce it on bug-gdb, or send them directly to the GDB maintainers at gdb-patches@cygnus.com.

23 Host Conditionals

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

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

`BLOCK_ADDRESS_FUNCTION_RELATIVE`
: dbxread.c

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

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

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

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

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

`ADDITIONAL_OPTIONS`
: main.c

`ADDITIONAL_OPTION_CASES`
: main.c

`ADDITIONAL_OPTION_HANDLER`
: main.c

`ADDITIONAL_OPTION_HELP`
: main.c

`AIX_BUGGY_PTRACE_CONTINUE`
: infptrace.c

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

`CFRONT_PRODUCER`
: dwarfread.c

`DBX_PARM_SYMBOL_CLASS`
: stabsread.c

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

`DEV_TTY`
: symmisc.c

`DO_REGISTERS_INFO`
: infcmd.c

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

ANSI definition.

`FILES_INFO_HOOK`
: target.c

`FLOAT_INFO`
: infcmd.c

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

`GCC2_COMPILED_FLAG_SYMBOL`
: dbxread.c

`GCC_COMPILED_FLAG_SYMBOL`
: dbxread.c

`GCC_MANGLE_BUG`
: symtab.c

`GCC_PRODUCER`
: dwarfread.c

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

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

`HAVE_TERMIO`
: inflow.c

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

`INT_MAX`

`INT_MIN`

`LONG_MAX`

`UINT_MAX`

`ULONG_MAX`
: Values for host-side constants.

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

`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

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

`CC_HAS_LONG_LONG`
: Define this if the host C compiler supports "long long". This will be defined automatically if GNU CC is used to compile GDB.

`PRINTF_HAS_LONG_LONG`
: Define this if the host can handle printing of long long integers via a format directive "ll".

`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, should be replaced by SEEK_SET instead, which is the POSIX equivalent.

`MAINTENANCE_CMDS`
: If the value of this is 1, then a number of optional maintenance commands are compiled in.

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

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

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

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

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

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

`NOTICE_SIGNAL_HANDLING_CHANGE`
: infrun.c

`NO_HIF_SUPPORT`
: remote-mm.c

`NO_JOB_CONTROL`
: signals.h

`NO_MMALLOC`
: GDB will use the `mmalloc` library for memory allocation for symbol reading, unless this symbol is defined. Define it on systems on which `mmalloc` does not work for some reason. One example is the DECstation, where its RPC library can't cope with our redefinition of `malloc` to call `mmalloc`. When defining `NO_MMALLOC`, you will also have to override the setting of `MMALLOC_LIB` to empty,

in the Makefile. Therefore, this define is usually set on the command line by overriding `MMALLOC_DISABLE` in '`config/*/*.mh`', rather than by defining it in '`xm-*.h`'.

`NO_MMALLOC_CHECK`
: Define this if you are using `mmalloc`, but don't want the overhead of checking the heap with `mmcheck`.

`NO_SIGINTERRUPT`
: remote-adapt.c

`NUMERIC_REG_NAMES`
: mips-tdep.c

`N_SETV`
: dbxread.c

`N_SET_MAGIC`
: hppabsd-tdep.c

`ONE_PROCESS_WRITETEXT`
: breakpoint.c

`O_BINARY`
: exec.c

`O_RDONLY`
: xm-ultra3.h

`PCC_SOL_BROKEN`
: dbxread.c

`PC_LOAD_SEGMENT`
: stack.c

`PRINT_RANDOM_SIGNAL`
: infcmd.c

`PRINT_REGISTER_HOOK`
: infcmd.c

`PROCESS_LINENUMBER_HOOK`
: buildsym.c

`PROLOGUE_FIRSTLINE_OVERLAP`
: infrun.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

`REG_STACK_SEGMENT`
exec.c

`REG_STRUCT_HAS_ADDR`
findvar.c

`R_FP` dwarfread.c

`R_OK` xm-altos.h

`SEEK_END` state.c

`SEEK_SET` state.c

`SEM` coffread.c

`SHELL_COMMAND_CONCAT`
infrun.c

`SHELL_FILE`
infrun.c

`SHIFT_INST_REGS`
breakpoint.c

`SIGTRAP_STOP_AFTER_LOAD`
infrun.c

`STACK_ALIGN`
valops.c

`STOP_SIGNAL`
main.c

`SUN4_COMPILER_FEATURE`
infrun.c

`SUN_FIXED_LBRAC_BUG`
dbxread.c

`SVR4_SHARED_LIBS`
solib.c

`SYMBOL_RELOADING_DEFAULT`
symfile.c

`TIOCGETC` inflow.c

`TIOCGLTC` inflow.c

`TIOCGPGRP`
inflow.c

`TIOCLGET` inflow.c

`TIOCLSET` inflow.c

`TIOCNOTTY`
inflow.c

`UPAGES` altos-xdep.c

`USE_O_NOCTTY`
inflow.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.)

`WRS_ORIG` remote-vx.c

`alloca` defs.h

`const` defs.h

`lint` Define this to help lint in some stupid way.

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

Platform-specific host conditionals.

`ALTOS` altos-xdep.c

`ALTOS_AS` xm-altos.h

`MOTOROLA` xm-altos.h

`NBPG` altos-xdep.c

`BCS` tm-delta88.h

`DELTA88` m88k-xdep.c

`DGUX` m88k-xdep.c

`F_OK` xm-ultra3.h

Regex conditionals.

`C_ALLOCA` regex.c

`NFAILURES`
regex.c

`RE_NREGS` regex.h

`SIGN_EXTEND_CHAR`
regex.c

`SWITCH_ENUM_BUG`
regex.c

`SYNTAX_TABLE`
regex.c

`Sword` regex.c

`sparc` regex.c

`test` regex.c

24 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.

`BLOCK_ADDRESS_FUNCTION_RELATIVE`
: dbxread.c

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

`ALIGN_STACK_ON_STARTUP`
: main.c

`ALTOS`
: altos-xdep.c

`ALTOS_AS`
: xm-altos.h

`BCS`
: tm-delta88.h

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

`BELIEVE_PCC_PROMOTION`
: coffread.c

`BELIEVE_PCC_PROMOTION_TYPE`
: stabsread.c

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

`BLOCK_ADDRESS_ABSOLUTE`
: dbxread.c

`BREAKPOINT`
: tm-m68k.h

`CALL_DUMMY`
: valops.c

`CALL_DUMMY_LOCATION`
: inferior.h

`CALL_DUMMY_STACK_ADJUST`
: valops.c

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

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

`CFRONT_PRODUCER`
: dwarfread.c

`DO_DEFERRED_STORES`

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

  Currently only implemented correctly for native Sparc configurations?

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

`DBX_PARM_SYMBOL_CLASS`
: stabsread.c

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

`DECR_PC_AFTER_HW_BREAK`
: Similarly, for hardware breakpoints.

`DELTA88`
: m88k-xdep.c

`DEV_TTY`
: symmisc.c

`DGUX`
: m88k-xdep.c

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

`DO_REGISTERS_INFO`
: infcmd.c

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

`EXTRACT_RETURN_VALUE`
: tm-m68k.h

`EXTRACT_STRUCT_VALUE_ADDRESS`
: values.c

`EXTRA_FRAME_INFO`
: If defined, this must be a list of slots that may be inserted into the `frame_info` structure defined in `frame.h`.

`EXTRA_SYMTAB_INFO`
: If defined, this must be a list of slots that may be inserted into the `symtab` structure defined in `symtab.h`.

`FILES_INFO_HOOK`
: target.c

`FLOAT_INFO`
: infcmd.c

`FP0_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`
: Given FRAME, return a pointer to the calling frame.

`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 (val, fi)`
: For the frame described by fi, set val to the number of arguments that are being passed.

`FRAME_SPECIFICATION_DYADIC`
: stack.c

`FRAME_SAVED_PC`
: Given FRAME, return the pc saved there. That is, the return address.

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

`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`
: Define this if you need to supply your own definition for the function `get_saved_register`. Currently this is only done for the a29k.

`GPLUS_PRODUCER`
: dwarfread.c

`GR64_REGNUM`
: Very a29k-specific.

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

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

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

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

`IGNORE_SYMBOL type`
: This seems to be no longer used.

`INIT_EXTRA_FRAME_INFO (fromleaf, fci)`
: If defined, this should be a C expression or statement that fills in the `EXTRA_FRAME_INFO` slots of the given frame *fci*.

`INIT_EXTRA_SYMTAB_INFO`
: symfile.c

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

`INNER_THAN`
: Define this to be either `<` if the target's stack grows downward in memory, or `>` is the stack grows upwards.

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

`SIGTRAMP_START`
`SIGTRAMP_END`
: Define these to be the start and end address of the sigtramp. These will be used if defined, and `IN_SIGTRAMP` is not; otherwise the name of the sigtramp will be assumed to be `_sigtramp`.

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

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

`KERNEL_DEBUGGING`
: tm-ultra3.h

`LCC_PRODUCER`
: dwarfread.c

`L_LNNO32`
: coffread.c

`MIPSEL`
: mips-tdep.c

`MOTOROLA`
: xm-altos.h

`NBPG`
: altos-xdep.c

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

`NOTICE_SIGNAL_HANDLING_CHANGE`
: infrun.c

`NO_HIF_SUPPORT`
: remote-mm.c

`NO_SIGINTERRUPT`
: remote-adapt.c

`NO_SINGLE_STEP`
: Define this if the target does not support single-stepping. If this is defined, you must supply, in `*-tdep.c`, the function `single_step`, which takes a pid

as argument and returns nothing. It must insert breakpoints at each possible destinations of the next instruction. See `sparc-tdep.c` and `rs6000-tdep.c` for examples.

`NUMERIC_REG_NAMES`
: mips-tdep.c

`N_SETV`
: dbxread.c

`N_SET_MAGIC`
: hppabsd-tdep.c

`ONE_PROCESS_WRITETEXT`
: breakpoint.c

`PCC_SOL_BROKEN`
: dbxread.c

`PC_IN_CALL_DUMMY`
: inferior.h

`PC_LOAD_SEGMENT`
: stack.c

`PC_REGNUM`
: If the program counter is kept in a register, then define this macro to be the number of that register. This need be defined only if `TARGET_WRITE_PC` is not defined.

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

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

`PRINT_RANDOM_SIGNAL`
: infcmd.c

`PRINT_REGISTER_HOOK`
: infcmd.c

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

`PROCESS_LINENUMBER_HOOK`
: buildsym.c

`PROLOGUE_FIRSTLINE_OVERLAP`
: infrun.c

`PS_REGNUM`
: parse.c

`PUSH_ARGUMENTS`
valops.c

`REGISTER_BYTES`
remote.c

`REGISTER_NAMES`
Define this to expand into an initializer of an array of strings. Each string is the name of a register. [more detail]

`REG_STACK_SEGMENT`
exec.c

`REG_STRUCT_HAS_ADDR`
findvar.c

`R_FP` dwarfread.c

`R_OK` xm-altos.h

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

`SEEK_END` state.c

`SEEK_SET` state.c

`SEM` coffread.c

`SHELL_COMMAND_CONCAT`
infrun.c

`SHELL_FILE`
infrun.c

`SHIFT_INST_REGS`
breakpoint.c

`SIGTRAP_STOP_AFTER_LOAD`
infrun.c

`SKIP_PROLOGUE`
A C statement that advances the PC across any function entry prologue instructions so as to reach "real" code.

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

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

`SP_REGNUM`
: parse.c

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

`STACK_ALIGN`
: valops.c

`STOP_SIGNAL`
: main.c

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

`SUN4_COMPILER_FEATURE`
: infrun.c

`SUN_FIXED_LBRAC_BUG`
: dbxread.c

`SVR4_SHARED_LIBS`
: solib.c

`SYMBOL_RELOADING_DEFAULT`
: symfile.c

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

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

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

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

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

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

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

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

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

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

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

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

`TARGET_READ_PC`

`TARGET_WRITE_PC (val, pid)`

`TARGET_READ_SP`

`TARGET_WRITE_SP`

`TARGET_READ_FP`

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

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

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

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

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

`WRS_ORIG`
: remote-vx.c

`test`
: (Define this to enable testing code in regex.c.)

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`.

25 Native Conditionals

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

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

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

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

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

`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`'.

`PTRACE_FP_BUG`
: mach386-xdep.c

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

`REGISTER_U_ADDR`
: Defines the offset of the registers in the "u area"; see Chapter 6 [Host], page 5.

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

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

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

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

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

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

`CLEAR_SOLIB`
: objfiles.c

`DEBUG_PTRACE`
: Define this to debug ptrace calls.

26 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!

27 The XCOFF Object File Format

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*, and search for XCOFF.

The shared library scheme has a nice 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).

Table of Contents