Working in GDB

A guide to the internals of the GNU debugger

John Gilmore
Cygnus Support

Cygnus Support
Revision: 1.28
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 1. Further work can be divided into "host-dependent" (see Chapter 4 [Adding a New Host], page 2) and "target-dependent" (see Chapter 5 [Adding a New Target], page 4). 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; unfortunately, that means you must add *both* host and target support for new machines in this category.

The `config/mh-*`, `xm-*.h` and `*-xdep.c` files are for host support. Similarly, the `config/mt-*`, `tm-*.h` and `*-tdep.c` files are for target support. The question is, what features or aspects of a debugging or cross-debugging environment are considered to be "host" support?

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.

Unix child process support is considered an aspect of the host. Since when you fork on the host you are still on the host, the various macros needed for finding the registers in the upage, running `ptrace`, and such are all in the host-dependent files.

This is still somewhat of a grey area; I (John Gilmore) didn't do the `xm-*` and `tm-*` split for gdb (it was done by Jim Kingdon) so I have had to figure out the grounds on which it was split, and make my own choices as I evolve it. I have moved many things out of the xdep files actually, partly as a result of BFD and partly by removing duplicated code.

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 GDB.

Then edit `bfd/configure.in`. Add shell script code 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.

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

(If you have the binary utilities and/or GNU ld in the same tree, you'll also have to edit `binutils/configure.in` or `ld/configure.in` to match what you've done in the `bfd` directory.)

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`'.

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- 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 1), there are two 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 5 [Adding a New Target], 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 some machines it doesn't exist at all.

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

Generic Host Support Files

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*-xdep.c.

coredep.c::register_addr()
: If your xm-*xxx*.h file defines the macro REGISTER_U_ADDR(reg) to be the offset within the 'user' struct of a register (represented as a GDB register number), 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*-xdep.c file, and be sure *xxx*-xdep.o is in the XDEPFILES 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.

Object files needed when the target system is an *xxx* are listed in the file config/mt-*xxx*, in the makefile macro 'TDEPFILES = '. . . . The header file that defines the target system should be called tm-*xxx*.h, and should be specified as the value of 'TM_FILE' in config/mt-*xxx*. You can also define 'TM_CFLAGS', 'TM_CLIBS', and 'TM_CDEPS' in there; see Makefile.in.

Now, you are now ready to try configuring GDB to compile for your system. 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
```

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

When hosting GDB 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-dep.c`; if not, it's in your *xxx*`-xdep.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.)

5 Adding a New Target

For a new target called *ttt*, first specify the configuration as described in Chapter 3 [Adding a New Configuration], page 1. 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/mt-`*ttt*
: 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`', 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.)

6 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!

7 Configuring GDB for Release

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

```
make 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.

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.

8 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.

9 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.

10 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).

11 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.

12 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.

Table of Contents