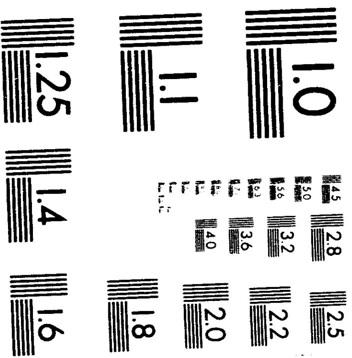
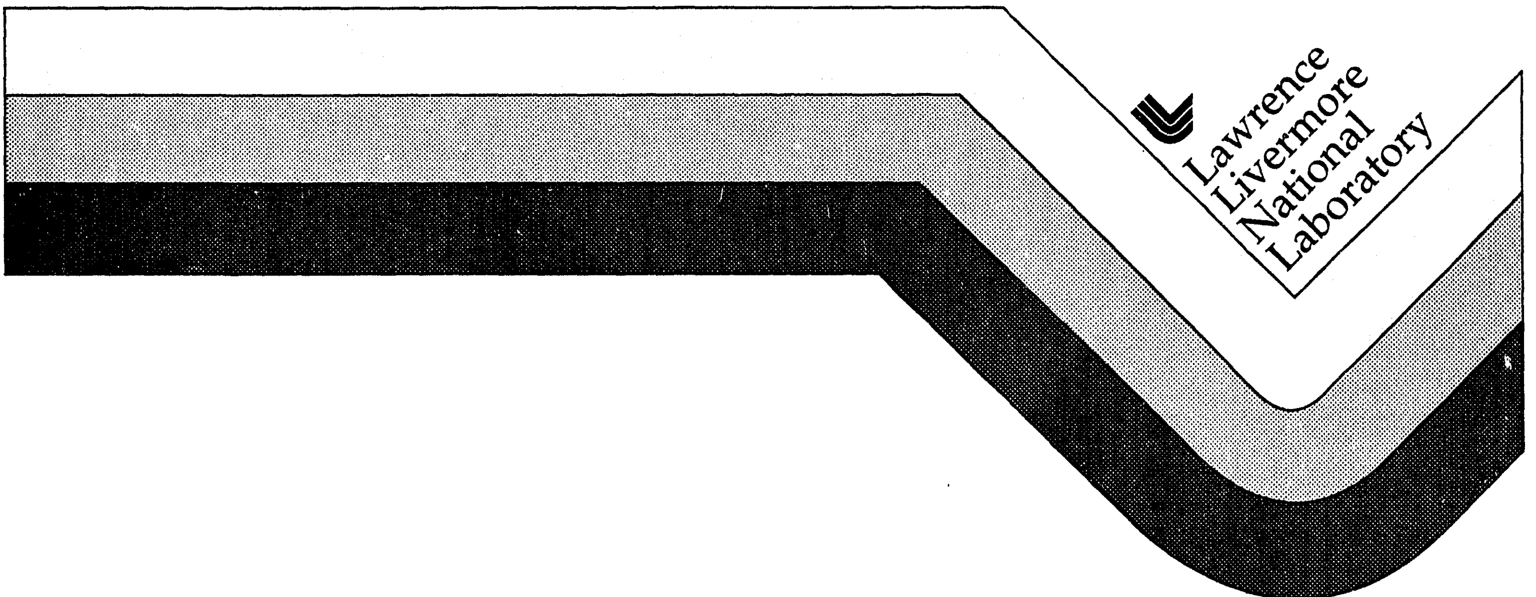


UCRL-MA-115656

GNU Debugger Internal Architecture

Robert Pizzi

December 16, 1993



DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government thereof, and shall not be used for advertising or product endorsement purposes.

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

GNU Debugger Internal Architecture

GNU gdb Internal Implementations
Edition 1.0 for gdb version 4.9 or earlier
December 1993

Abstract:
This document describes the internal architecture and implementation of the GNU debugger, gdb. Topics include inferior process management, command execution, symbol table management and remote debugging. Call graphs for specific functions are supplied. This document is not a complete description but offers a developer an overview which is the place to start before modification.

Robert Pizzi
(rpizzi@llnl.gov)
Department of Applied Science
University of California Davis
Lawrence Livermore National Laboratory

December 16, 1993

MASTER

Edited by:
Dr. Patrick Miller & Dr. Dan Nessett
Lawrence Livermore National Laboratory

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Table of Contents

1 Introduction 1
1.1 Why Modify GDB? 1
1.2 Design Overview 1
1.3 Scope 2

2 Command Architecture 3
2.1 Command Types 3
2.2 Calling Structure 3
2.3 struct cmd_list_element 4
2.3.1 Command Type 5
2.3.2 Command Class 5
2.3.3 Remaining Fields 6
2.4 Command Organization 7

3 Debug Target Process 8
3.1 Debugging Target 8
3.2 Debug Tasks 9
3.2.1 File Command 9
3.2.2 Run Command 9
3.2.3 Next/Step Commands 10

4 Symbol Table Management 12
4.1 Objfile Structure 12
4.1.1 Global Access 12
4.2 Symbol Functions 13
4.2.1 Symbol Function Structure 13
4.2.2 Objfile and sym_fn Relationship 14
4.2.3 BFD Usage Through Symbol Readers 15
4.3 Symbols and Symbol Tables 15
4.3.1 Symbol Table Organization and Types 16
4.3.1.1 Symtab Details 17
4.3.1.2 Psymtab Details 19
4.3.1.3 Minimal Symbol Table 20
4.3.2 Symbols 21

5 Remote Debugging Model 22
5.1 Remote Stub 22

5.2 Communications Model 22
5.3 Source Files 23

6 Binary File Descriptor Library 24

6.1 BFD Top Level Architecture 24
6.1.1 User - BFD Interface 24
6.1.2 BFD Library - Object File Interface 25
6.2 BFD Internals 25
6.2.1 BFD Front End 26
6.2.2 BFD Library Code 26
6.2.3 BFD Back End 27
6.3 BFD Data Structure 28
6.3.1 Object File Data 28
6.3.2 Private Store 28
6.4 Obstack 29

7 Summary 31

Appendix A Source File Summary 32

A.1 Source File Summary 32
A.2 Remote Debugging Routines 33
A.3 Target Dependent Codes 33
A.4 GDB Source Tree 33

Appendix B Call Graphs 35

B.1 Symtab Graph 36
B.2 Command Graph 36

1 Introduction

This report is intended for the programmer who wishes to enhance the GNU debugger, GDB, version 4.9. It describes the major design features of the debugger. The information in this report was gathered from the documentation supplied by the GNU project in the GDB release and from direct examination of the source code.

This document and call graphs mosaics (see Appendix B [Call Graphs], page 35) are available via anonymous ftp from `sisal.llnl.gov (128.115.19.65)` in the `pub/gdbDocument` directory. This directory contains this document in postscript and TeX forms. The call graphs are only available in postscipt form.

1.1 Why Modify GDB?

The Free Software Foundation philosophy of software design encourages programmers to enhance their products. This is done by allowing programmers free access to the source code. In the case of GDB, this philosophy allows programmers to provide support for the multitude of platforms and to customize the program for a specific architecture. As new platforms are introduced, programmers write code that fits into the GDB design, thus extending the debuggers portability. Portability is a primary goal of all GNU software. Of course, there is a price for allowing portability. Complex data structures are required to support multiple platforms. It is a goal of this document to provide a measure of guidance in describing data structures used by GDB.

The underlying principle that guides all designs within GDB is that the debugger must be able to run on and be able to debug a significant number of machine architectures. This generality requires that the implementation have 'kludges' to account for such things as compiler bugs, O/S differences, and inherent differences between microprocessors. Although, this report does not detail these differences, when examining the source code, the comments proceeding the kludge usually explain its purpose. When modifying the source code, it is the programmer's responsibility to understand the kludges so that portability is not lost as changes are made.

1.2 Design Overview

Within the 20Mb of source files in the GDB release, two general categories of files can be defined. The first category includes 'core' source files. Core source files define the overall implementation of the debugger, such as the command parser and symbol table readers. The design of the core routines is not specific to any platform or architecture. The second category is that of machine dependent

files. These files are only compiled and linked into GDB when compiling for that particular machine. Machine dependent files are not detailed herein, except as to how they fit into the overall design of the debugger. For a more detailed look at the source files, see Section A.1 [Source File Summary], page 32.

The core source files are further organized into smaller, distinct parts. The parts include:

- Debugger Command Parser
- Debug Process Management
- Object File Symbol Format Reading
- Symbol Table Management
- Remote Debugging

The *Debugger Command Parser* organizes the commands of the debugger and presents a usable interface to the user. *Debug Process Management* is primarily concerned with controlling the program the user wishes to debug. *Object File Symbol Format Reading* allows the debugger to parse an executable file to obtain debugging symbol information. On single platform debuggers, this is trivial, since that debugger will only support one object file format. Since GDB is a multi-platform debugger, it has to support numerous object file formats such as *COFF*, *ELF*, *a.out*, *IEEE* and *Oasys*. The *Symbol Table Management* must be able to organize the huge number of symbols generated in an object file, and be able to access the data the symbols represents within the context of the program being debugged in a relatively machine independant way. Finally, *Remote Debugging* provides a method where GDB can debug programs on machines that have a limited capability which cannot support a full debugger, such as an embedded system.

The following chapters will describe each of the above core parts, giving overall designs, and data structures used to support them.

1.3 Scope

This report does not detail every aspect of the GNU debugger, rather it covers only those topics discussed above. See Section 1.2 [Design Overview], page 1. A significant part of the debugger that is not covered is the methodolgy used to show source code, show variable values, set variable values, and setting the various types of breakpoints. The justification for omitting those parts from this report is that the implementations use the parts that are discussed. Thus, more inforamation could be gained by directly examining the source code for those implementations while using this report as a reference to the data structures.

2 Command Architecture

The purpose of the Command Architecture is to provide a flexible way to define commands that effect the state of the debugger and that of the debugged process. This design has been influenced by the requirements of supporting multiple platforms and an extensible command set. At run-time, any source module may add its own commands to the debugger as needed to support special features. This chapter describes the composition of a command, and how different 'types' of commands are organized and executed by the debugger.

2.1 Command Types

Commands within the debugger are organized into 'types' or 'categories'. 'Core' commands are those that effect the general state of the debugger and that are always available regardless of the platform. For example, `list`, `break` and `run` are all core commands. Another type of command describes those that are platform dependent. For example, architecture specific code may add a command to access special registers. The third type of command is one that is defined by the user as a sequence of commands. It is important to note that the type of command does not influence how a command is executed. For more details about the different command types, see Section 2.3.1 [Command Type], page 5.

2.2 Calling Structure

The Figure 1 below, Command Execution Call Graph, is a call graph of the execution of the debugger when a command is entered by the user. After the initialization of the debugger in the '`main.c`' code, the '`command_loop`' routine is executed. This routine implements the user interface.[1]

The user interface provides support for entering commands from '`stdin`' with features such as line editing, command repetition, and partial command completion. Once a valid command has been entered, the '`execute_command`' routine is called.

[1] The current supported interface is a command line. An X Windows interface, called 'xxgdb' exists, but GNU recommends using GNU Emacs in gdb mode. GNU does not support xxgdb.

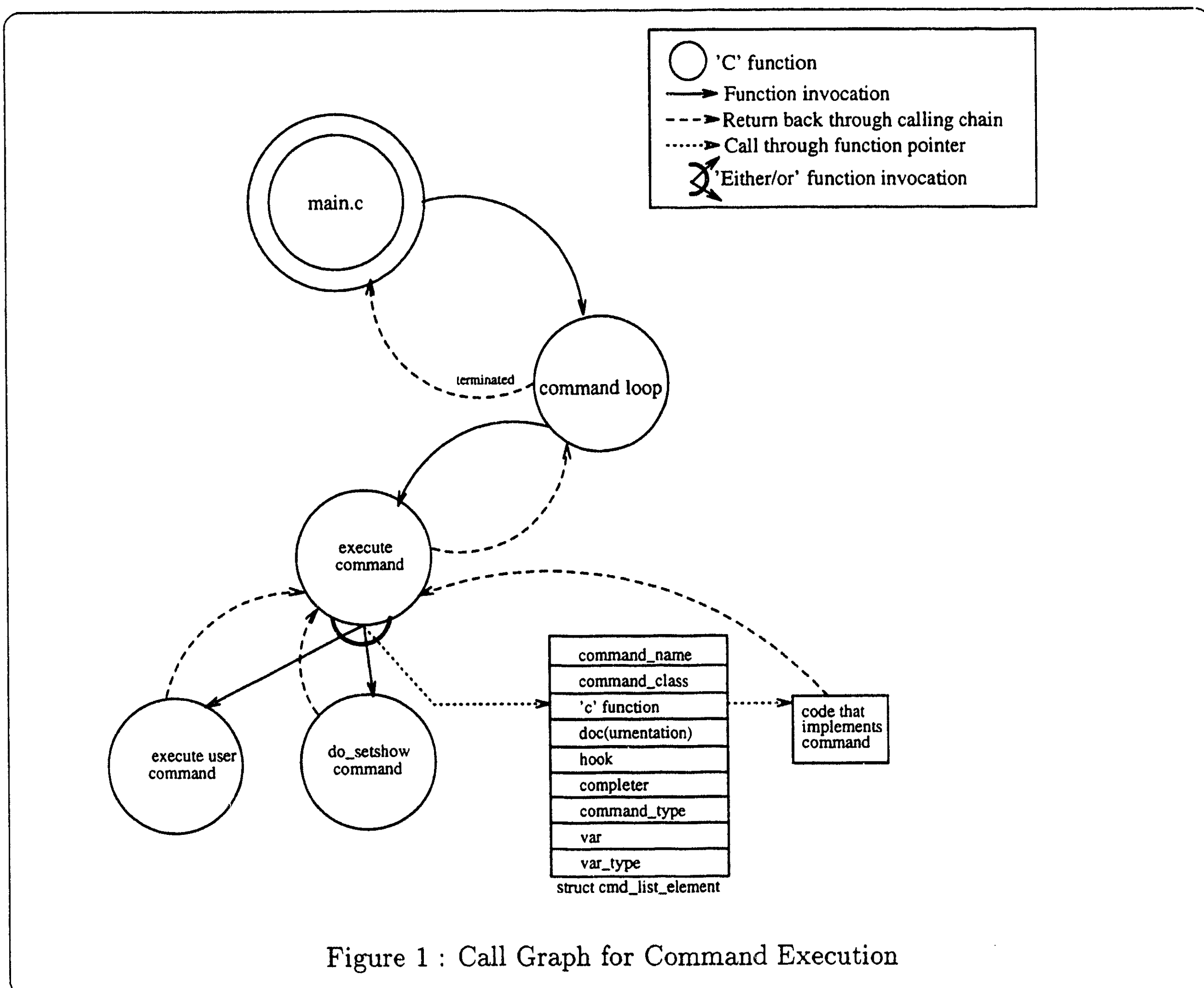


Figure 1 : Call Graph for Command Execution

After a user enters a command from the command line, the Execute_command routine looks up the command in gdb's linked list of defined commands. If found, a canonical data structure (see Section 2.3 [struct cmd'list'element], page 4) is returned that defines the action of the command. The action of a command depends on the command type. If the command is a user-defined type, or a 'set' command type, then a routine specifically implemented to handle all user or set commands is called directly. All other commands are executed by indirectly calling a routine through a jump vector to code that performs the actions required for the command. When the command is finished, control returns to the command loop.

2.3 struct cmd_list_element

The `cmd_list_element` (command list element) is the canonical data structure that defines how a GDB command executes. Figure 1 shows a partial listing of the fields within the structure. In addition to the command type, this structure maintains a command class field which further

defines a command's action. Also, fields exist which allow patching a command's action with machine dependent code.

2.3.1 Command Type

A command type is a classification of the type of action a command performs. The possible types are `set_cmd, show_cmd, not_set_cmd`. (Set command, show command, not set command, respectively.) A `set_cmd` type describes commands that have a value, and can be changed by the user. A `show_cmd` type describes commands that display the value of an attribute to the user, such as '`show radix`'. Commands that are neither set nor show commands are `not_set_cmd`, such as '`run`'.

When a command of type `set_cmd` or `show_cmd` is parsed, the routine '`do_setshow_command`' is executed. This routine obtains a `cmd_list_element` from the command loop that represents the debugger attribute. For example, in the command '`set prompt`', the '`set`' indicates to the command loop that this is a set command; thus, the keyword that follows indicates an attribute. This attribute is searched for, and if found, its `cmd_list_element` is passed to the '`do_setshow_command`' routine.

The command list element for a set/show command contains two fields that describe the attribute. The first field is labeled `var`. This is implemented as a `char *` variable, but it is generally used as a pointer to data of whatever type the attribute maintains. The second field is labeled `var_type`. This is an enumeration that specifies the type of data the `var` pointer references. The '`do_setshow_command`' then examines the attribute type and manipulates the data based on type. The `var_type` enumeration specifies string, integer, boolean, and filename parameters.

When a command is of type `not_set_cmd`, then the command loop expects to find a jump vector to code that implements the function. The jump vector is a function pointer, labeled `cfunc` ('C' function) in the command list element structure. Control is passed to the specified routine, passing it the remainder of the command line to parse, if needed.

2.3.2 Command Class

The command class is an enumeration that refines the specification of a command beyond the scope of a command type. The command parser examines this field to discriminate between commands that are of type `not_set_cmd`. For example, the '`run`' command is of type: `not_set_cmd`; class: `class_run`, while the '`list`' command is of type: `not_set_cmd`; class: `class_files`.

There is a special class called `class_user` which allows users to define their own commands. A class_user command is composed of existing debugger commands, such as shown below.

```
(gdb) define samplecommand
show prompt
echo running sample
run
end
(gdb)
```

The example defines a command, '`samplecommand`', which is composed of the other commands, '`show prompt, echo, run`'. When the command loop parses a user-defined command, control passes to the routine '`execute_user_command`'.

A user command consists of a linked list of *copies* of the command list element data structures for the commands defined by the user. The instantiation of the user command above would consist of a linked list of three command list elements, that represent the `set prompt, echo, run` commands. Thus, the '`execute_user_command`' routine simply calls the '`execute_command`' routine recursively[1] with each of the commands in the linked list.

2.3.3 Remaining Fields

The `command_name` fields points to an ASCII representation of the name of the command. This is used during command look-up. The `doc` field is a pointer to ASCII text that documents the command. This string is displayed when the '`help`' command is used.

The `hook` field is a pointer to a linked list of command list elements. This is used to store the commands of a user defined command. Also, for other commands, this pointer provides a way to execute other commands before executing the primary command. Using the hook in this manner is only good for 'patching' existing commands. For example, if the Sun architecture required any caches to be purged before debugging a process, then the Sun architecture specific code would add a hook to the '`run`' command such that it would execute another command, say '`flush`', before '`run`' is executed. '`Flush`' would be an architecture specific command added and implemented in the source files for the Sun platform.

[1] Forward references to commands are accepted. Thus, semantic checks on the commands are not done, until the user command is executed. Caution is advised, since a user can create a cycle, which will hang gdb.

The `completer` field is a jump vector to a routine that matches a command to a partially entered command from the user. All commands use the same completer if it can be partially identified. Commands with a `NULL` completer field imply that the command must be entered completely. For example, the `run` command may be entered as `ru`, since it has a completer.

2.4 Command Organization

GDB has a suite of routines that manage and manipulate the command list element structures.[2] When a command is created, (by calling the appropriate routine), the command structure is added to a global linked list of commands. This list is the one searched during command look-up. During initialization of GDB, each source module has an opportunity to have its initializer called. In this initializer, commands are added, and hooks are made to existing commands.

[2] Routines are defined in the `command.h` & `command.c`

3 Debug Target Process

One of the most important aspects of any debugger is that of control over the program to be debugged. This chapter describes how GDB controls its debugged program by forking a child process, controlling the child process, and stepping through execution of the program. The process forked by GDB is commonly referred to as the inferior child process, or just inferior.

The sections below discuss, at a high level, the tasks GDB performs while debugging a program. The tasks described are very architecture dependent as to their exact implementation. However, all implementations of the routines are conceptually the same.

3.1 Debugging Target

GDB is able to debug a variety of executable formats, such as `a.out` or a library archive. An abstraction is used to represent an executable. It is called a *target*.[1] The purpose of the target is to provide a specification of target dependent routines required for debugging. This specification is a structure containing function pointers. The structure's data fields represent abstract debugging functions, such as resuming, waiting, and fetching register values.[2]

There are four debugging targets defined: `none`, `core`, `exec`, `child`. The `none` target indicates no program is currently being debugged. Thus, all debugging functions specified are no-ops. The `core` target indicates that a core file is being used during the debugging session. Core files don't execute, but accesses to memory (i.e. showing a variable's value) are made through the core file's image rather than the inferior's memory image. Both the `exec` and `child` target refer to executable files. The difference between the two is that each represents a different 'phase' of the executable during debugging. An executable file that is not currently running is an `exec` target. An `exec` target allows access to source code such that it can be viewed (listed), and breakpoints (etc.) can be set. Access to variables is not allowed due to the fact the debugged program does not yet have a memory image.

Once the `run` command is used, the executable becomes a `child` target. A `child` target does everything an `exec` target does and allows access to variables and the state of the inferior. The executable remains a `child` target until it terminates. At that time, the executable reverts to an `exec` target.

---

[1] Note, this abstraction is not to be confused with the BFD Library's target, see Section 6.2.2 [BFD Library Code], page 26

[2] Examine the file `target.h` for definitions

3.2 Debug Tasks

This section describes the high level tasks used during debugging. Those tasks include specifying which program to debug, starting and stopping the inferior's execution, and stepping through the inferior's code. The tasks are diagrammed in Figure 2, Debug Tasks. In the figure, the 'execute_command' task only shows the commands relevant to this chapter. See Chapter 2 [Command Architecture], page 3.

3.2.1 File Command

The 'file command' task is initiated when the user wants a new executable file to debug, after GDB is already running (file command). Note, similar code is called if the user specified an executable when invoking GDB from the shell, but it is not the 'file_command' task. This task prepares and initializes GDB for debugging and loads partial symbol table information. See Section 4.3 [Symbols and Symbol Tables], page 15. After loading the symbols, the executable is designated as an exec target.

3.2.2 Run Command

The 'run_command' routine is initiated when the run command is used. First, the phase of the executable changes to a child target. Then, an inferior debugging process is created using the create and fork tasks. The inferior process then executes the UNIX command EXECLP. This allows the forked inferior to execute the program that is to be debugged.

The inferior process has been created such that it is in a 'trace' mode. GDB must wait until the EXECLP command stops before it has access the inferior's memory image. It stops at the first instruction of the debugged executable. GDB then uses a system call to proceed with the execution of the inferior.[3]

Once the child resumes execution, GDB waits for the inferior to do something. The debugger waits for *an event of interest*. This event can be a breakpoint or signal that stops the inferior. Note that GDB does not directly stop the inferior. The inferior stops itself because of the event and signals GDB that it has stopped.[4] When stopped, GDB will then execute the 'normal stop' task. If no such events occur, the inferior executes until the program completes normally.

[3] The system call is either ptrace or /proc if running SVR4. The interfaces for the two are very different. Fortunately, through the use of debugging targets, the implementations of the debugging functions are insulated, thus either ptrace or /proc are used transparently

[4] This signaling is generally done by GDB using wait()

3.2.3 Next/Step Commands

These distinct commands perform very similar tasks. The '`step command`' task executes the next instruction of the inferior. If that happens to be a function, then it 'steps' into the function and stops on the first instruction of the function. The '`next command`' task executes the next instruction, but if it happens to be a call, the function call is completed and the inferior stops on the next instruction following the call.[5]

[5] Provided no events of interest occur when executing the function call.

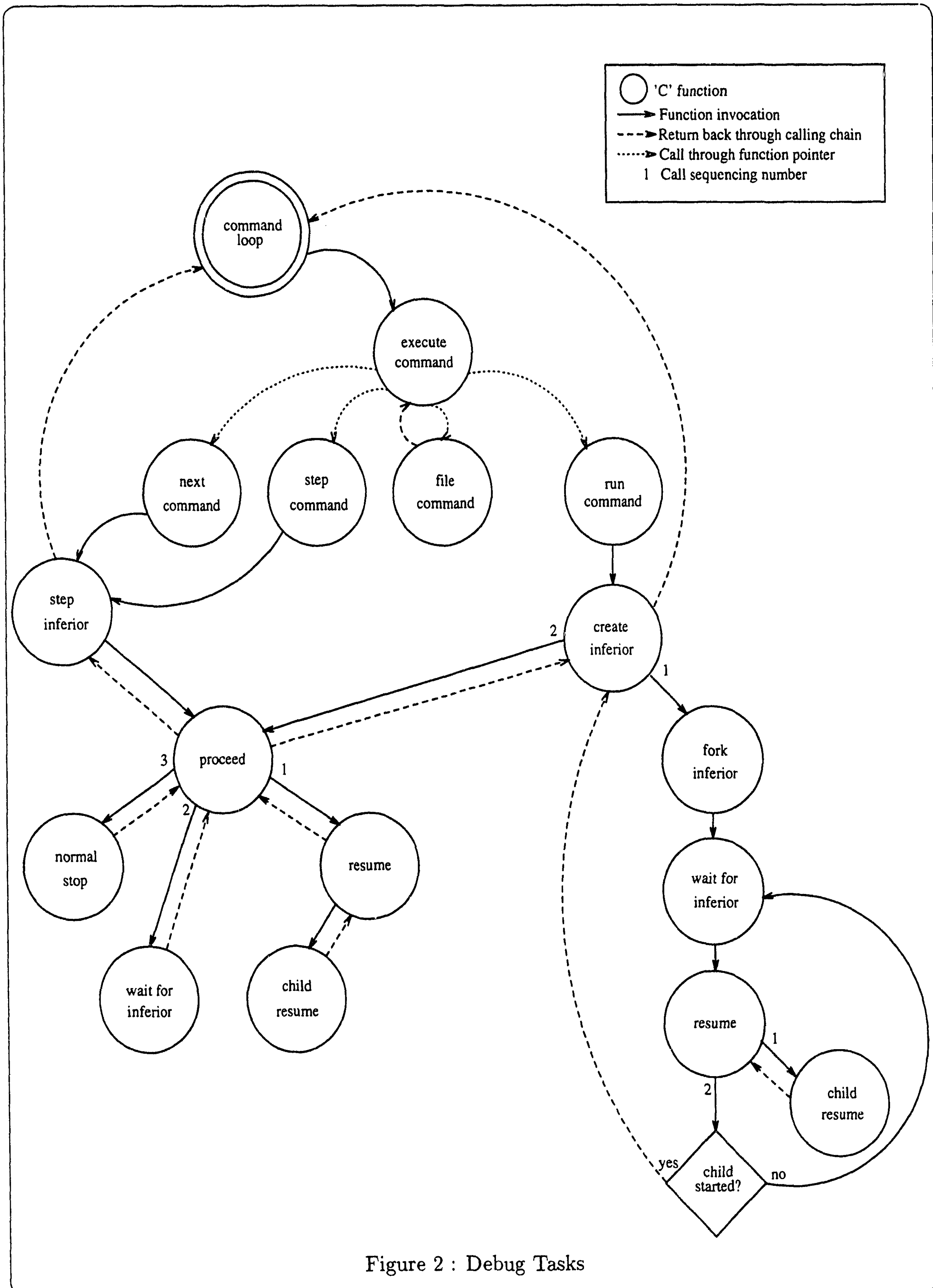


Figure 2 : Debug Tasks

4 Symbol Table Management

This chapter describes how GDB manages the symbols obtained from a program that a user is debugging. This data includes information pertaining to the executable file itself, source code, and symbol table entries. GDB uses a specially developed library called the Binary File Descriptor Library (BFD) to support the numerous executable file and symbol table foramts. This library provides a high level interface for accessing the data in an executable file, regardless of the format. Thus, GDB is able to efficiently support existing and future platforms by using the BFD Library. As a result, GDB is modular and expandable. See Chapter 6 [Binary File Descriptor], page 24, for more information on the BFD Library.

4.1 Objfile Structure

GDB uses an abstraction to manage the executable file currently being debugged. This abstraction represents the executable as a data structure named `struct objfile` (object file structure). There is a corresponding object file structure for every executable that GDB is currently debugging. Also, if a core file is used during debugging, it too has an object file structure. The purpose of this structure is to provide the starting point for describing the executable program using other GDB data structures such as the symbol tables.

The structure maintains the following data about the executable: filename, symbol tables, obstacks, symbol functions, and private data. The filename is the pathname of the executable. The obstacks are pools of free store for use by special memory management routines. See Section 6.4 [Obstack], page 29. Private data is used by platform specific routines to store information that the GDB structures do not account for. Symbol table, and the symbol functions are described below.

4.1.1 Global Access

All object file structures are maintained in a linked list which is globally accessible. Three global variables are used to maintain the list. The first is named `object_files` which is a pointer to the head of the linked list of object file structures. The second is named `current_objfile` and it is also a pointer; one that points to the object file that the user is currently accessing.[1] Finally, the third is named `symfile_objfile`, and it is also a pointer; one that points to the object file from which the main symbol table has been processed. In the typical case, all three variables point to the same object file structure, since only one is used. In general, care must be used when accessing these structures due to their global nature and lack of macros to access the data transparently.

[1] The user accesses the object file through use of GDB commands such as `list, run, print`...

4.2 Symbol Functions

Symbol functions are data structures that maintain information relating to a specific format of the symbols stored within an object file. GDB maintains a linked list of all supported symbol functions. This list is created during initialization by specific routines providing the required data. These special routines are grouped together into source files and are called *symbol readers*. Each type of symbol format has a distinct reader. The reader is responsible for providing a standardized interface that allows GDB to seamlessly access the object file's symbols, regardless of the format used to represent them in the file.

4.2.1 Symbol Function Structure

The data structure that represents the symbol functions is named `struct sym_fns`. The structure contains data that specifies information about a particular symbol reader, such as the name of the symbol format and jump vectors to routines within the symbol reader's code. These jump vectors provide the standard interface that the rest of GDB uses to access the symbols. The interface provides four routines. The first is called `sym_new_init`. It is used to provide first-time initialization of the symbol reader and process any data that is global to the entire symbol table. The routine is invoked only when an entirely new executable is to be debugged. The second routine is called `sym_init`. This routine is used to initialize private data in the symbol function structure for the reader and read in any additional data not processed in `sym_new_init`. This routine is called every time GDB needs to read a symbol table file (object file, archive, core) for any reason.

The third routine is the workhorse, `sym_read`. Its purpose is to read the object file's native symbols can convert it into a GDB symbol table. Finally, the routine `sym_finish` is required to perform necessary 'clean-up'. All of these routines must convert the symbol data from the format in the executable file to canonical data structures that GDB uses. See Section 4.3 [Symbols and Symbol Tables], page 15.

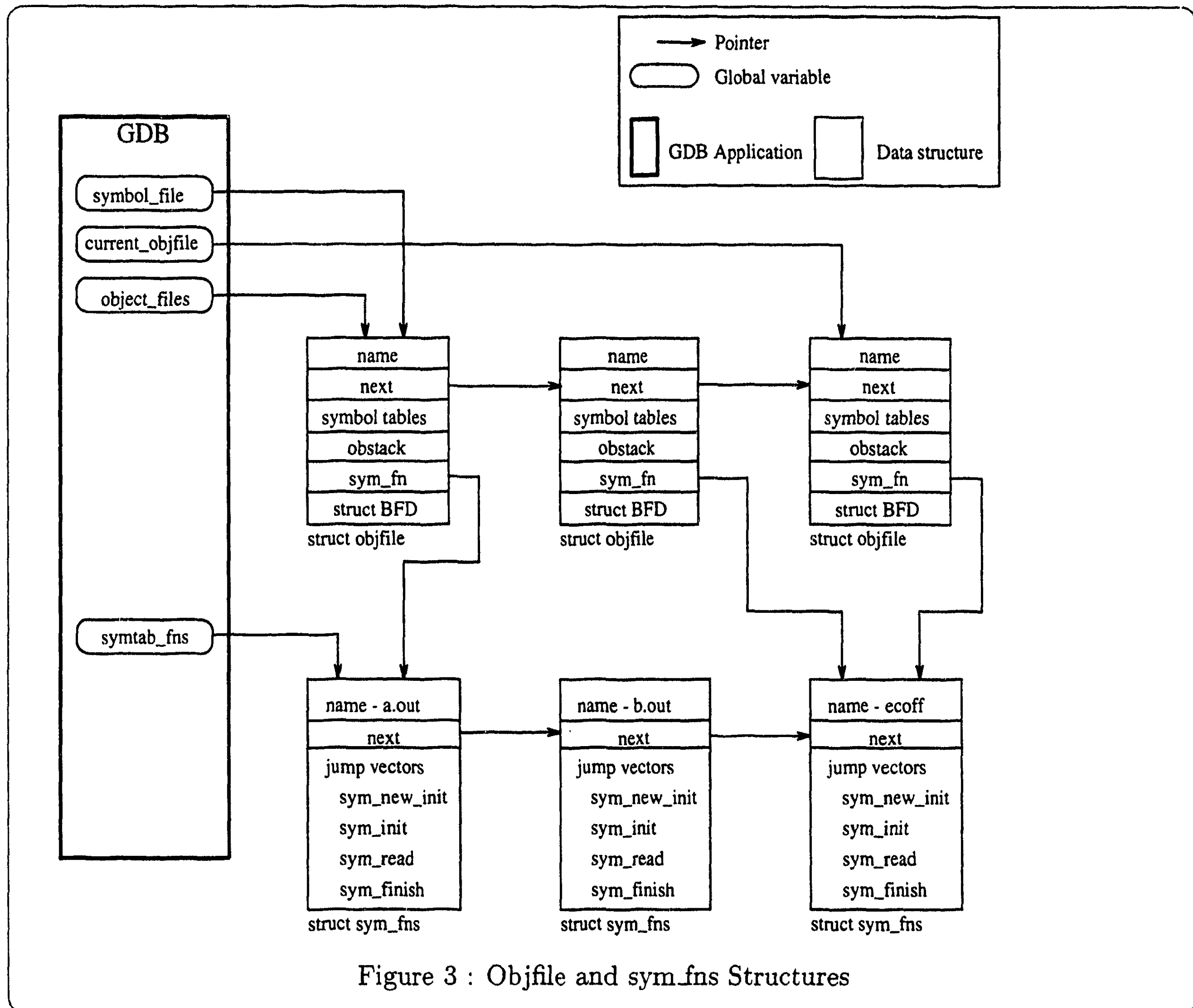


Figure 3 : Objfile and sym_fns Structures

4.2.2 Objfile and sym_fn Relationship

Figure 3 diagrams the relationship between the object file structure and the symbol functions. Both structures are maintained in a linked list accessed by global variables. When an object file is opened for debugging, an object file structure is created for it. During this opening, GDB searches through the list of symbol functions to find one that recognizes the symbol format of the object file. When one is found, a pointer to the symbol function structure is stored in the object file structure. Multiple references to symbol function structures are allowed.

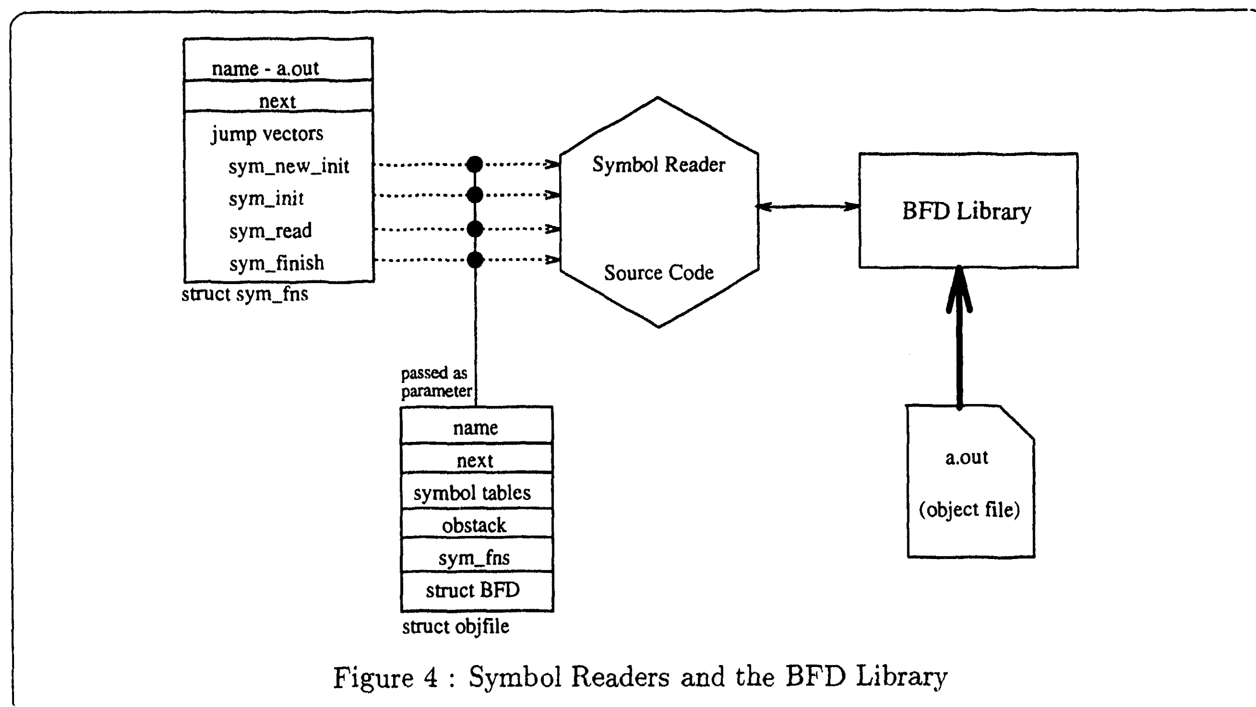


Figure 4 : Symbol Readers and the BFD Library

4.2.3 BFD Usage Through Symbol Readers

The BFD Library (see Chapter 6 [Binary File Descriptor], page 24) manages the interface between the symbol reader and the object file. See Figure 4. It is assumed that the symbol reader knows more about the symbol format that it is designed for than does the BFD Library. Thus, the symbol reader is free to use BFD in any way it sees fit to perform its tasks. This allows any particular symbol reader to kludge any incompatibilities between the native symbol format and the GDB canonical formats. For example, the `a.out` symbol reader uses low-level BFD routines to load the symbol string table, and store a reference to a block of memory allocated for it in the private pointer in the object file structure.

4.3 Symbols and Symbol Tables

An efficient compiler must have an efficient symbol table. Likewise, so will an efficient debugger. GDB has a complicated symbol management system. Unfortunately, some efficiency is sacrificed for the flexible design that supports many different native symbol formats (symbol format in the object file). GDB solves this problem by defining a set of canonical data structures that define a symbol within the debugger. It is the responsibility of platform specific routines (symbol readers) to convert the native symbol format to the canonical structures. Any data that cannot be canonized *and* is an important part of the native symbol table must be stored in private data structures and managed independently by platform specific routines.

4.3.1 Symbol Table Organization and Types

Within the object file structure are references to the symbol tables maintained in the debugger for the object file. There are three types of symbol tables used in GDB, symtab, psymtab and minimal. Thus there are three references within the object file structure. The first type of symbol table is the symtab (symbol table). A reference to a symtab points to a linked list of `struct symtab` instances. Each symtab instance represents complete symbol table information about all symbols within one source file that was compiled as part of the object file. See Figure 5.

The second type of symbol table is the psymtab (partial symbol table). A reference to a psymtab is an instance of a `struct psymtab`. This table consists of partial information about the symbols in a particular source file. The difference between the symtab and psymtab structures is exploited by each type of symbol reader. Creating a partial symbol table requires only a partial processing of the symbols in the object file. Thus, partial symbol tables may be created faster and are smaller than the full symbol tables. For most debugging commands, only the partial symbol table information is needed. When detailed symbol information is required, additional processing can be done on a file by file basis to obtain the symtab information. The third type of symbol table provides a minimal amount of data on the symbols within an object file. A reference to this table access an instance of a `struct minimal_symbol`. This table is not organized the symtab or psymtab. Only selected symbols are processed and placed in this table for each object file. The symbol reader decides which symbols to add to the table.

The various symbol readers implement the creation of associated symbol tables with varying degrees of processing effort. For minimal symbols, `struct minimal_symbol` lacks private data fields for the readers usage. However, both the psymtab and symtab structures have such fields. Thus, more information is kept in the partial symbol table allowing quick conversion to the symtab format. However, due to the increased complexity of the psymtab structure, creation will be slower than that of the minimal symbols. Likewise, if psymtab information is dropped, then conversion to the symtab format may take significantly longer then if the psymtab already had the data, since the object file will have to be re-processed. Whatever way the individual symbol readers decide to implement, there are trade-offs to be balanced. It is generally better, however, to provide quick creation of the partial symbol tables, since partial symbols are always created first when a new object file is debugged. Then, the conversion of the psymtab to a symtab will take more processing effort and time.

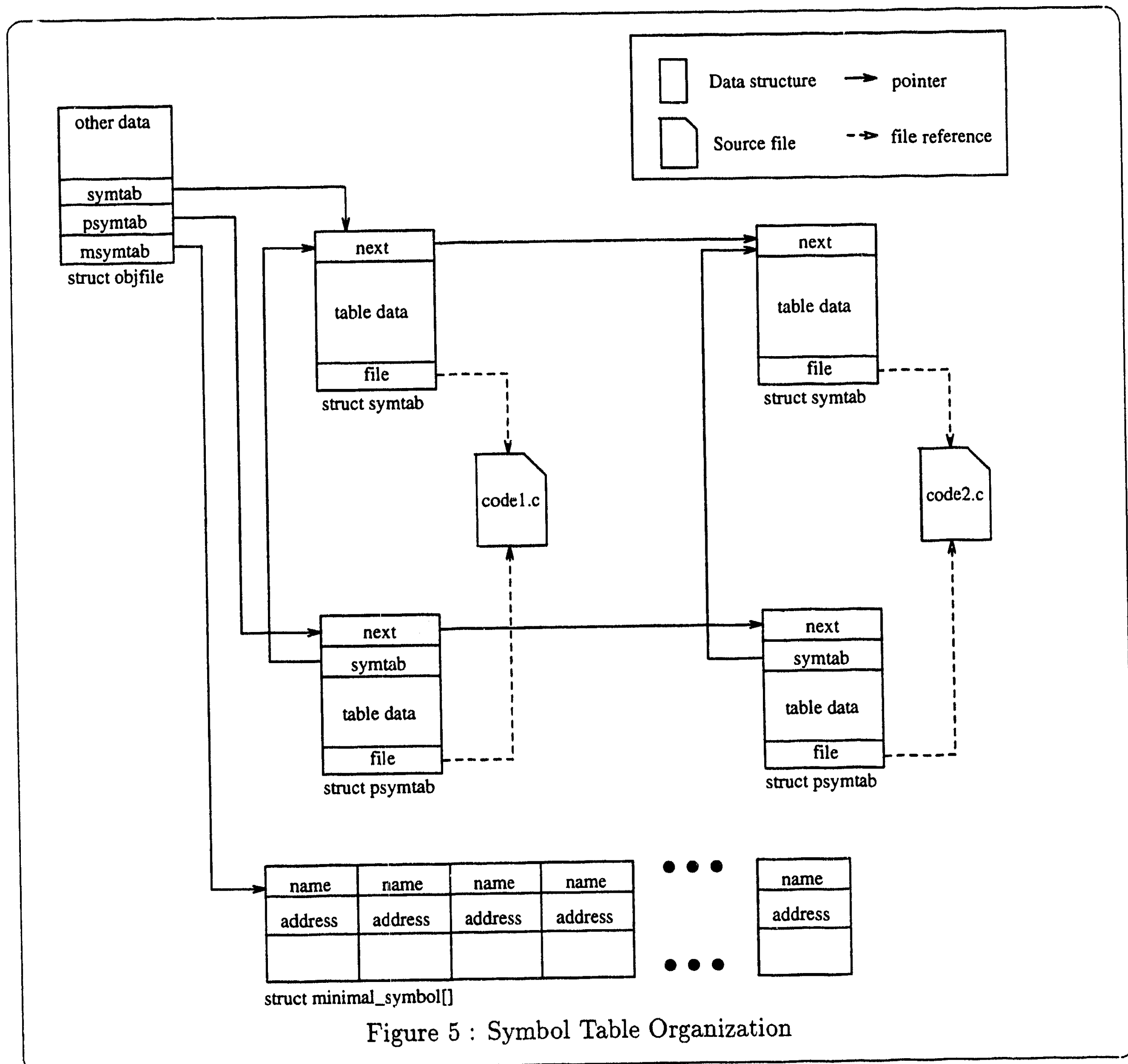


Figure 5 : Symbol Table Organization

4.3.1.1 Symtab Details

The symtab structure contains the most complete information about the symbols within an object file. A single instance of a symtab describes a single source file used in the compilation process of the object file.[1] Thus, to describe an object file, a linked list of symtab structures is used, one structure for each source file.

---

[1] Assuming that the object file was compiled with debugging symbols. Depending on the reader, even without debugging symbols, moderately useful symbol tables can be created.

Within the symtab structure is a significant amount of information about the symbols within a particular source file. It is the responsibility of the symbol reader to create these tables. The reader must then convert symbols in the native format in the object file to the symtab structure. If a reader must store data that cannot be canonized, it may do so by using private data pointers within the symtab structure. Figure 6 diagrams the symtab structure.

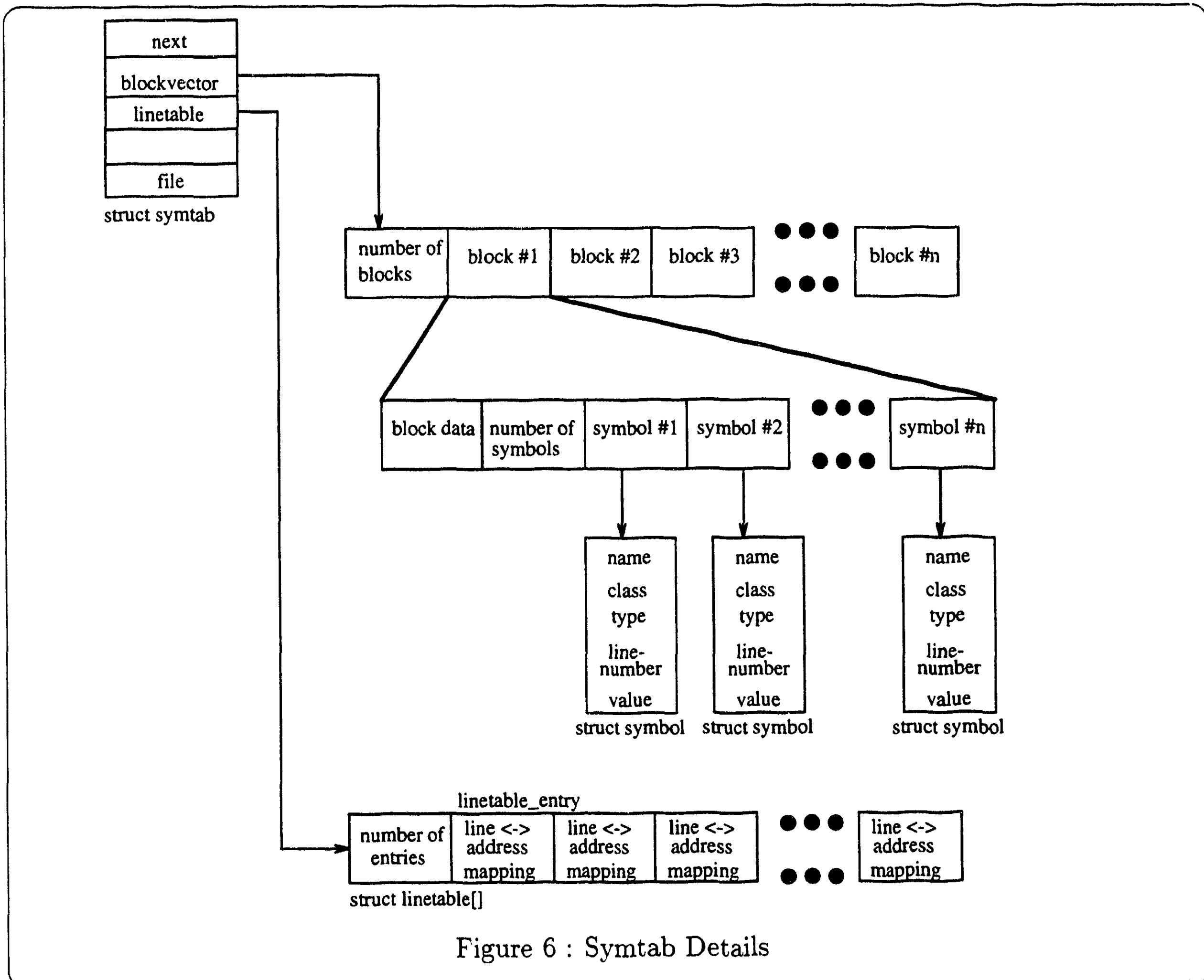


Figure 6 : Symtab Details

Figure 6 shows that the **blockvector** is a variable sized structure of blocks. A block represents a name-scope contour of the source file. Each block represents one name scope. Also, each lexical context is described by a block. Within the blockvector, the first two blocks are 'special'. The first block contains all symbols defined in the source file whose scope is the entire executable. The second block contains all symbols whose scope is the entire source file, but not the entire executable. These correspond to both 'C' global symbols and static symbols.

Each block also records a range of addresses for the code represented by the block. Thus, the 'special' blocks give the address range for all code in the source file since the scope of these blocks is the entire program. The blocks are implemented as a variable sized structure consisting of a fixed part and a variable number of pointer references. These references are to unique instances of `struct symbol`, Section 4.3.2 [Symbols], page 21.

The symtab structure also references another canonical structure. The `linetable` exists to provide a mapping of source code line numbers to object file addresses and vice versa. This table is usually large, but the size of individual entries in the table is never more that two pointer-sized variables. The table's entries are organized by the symbol reader and are usually sorted in ascending order by the address. Symtab structures are not initially created by GDB. Only when the debugger requires detailed information about one source file will it create this type of table.

4.3.1.2 Psymtab Details

The partial symbol table structure, `struct psymtab`, contains partial information about the symbols within a source file. This table is organized exactly like the symtabs. See Section 4.3.1.1 [Symtab Details], page 17. However, the data maintained in the partial symbol table differs from that of the data in the minimal and full symbol tables.

The data in the psymtab is minimal. It contains a set of jump vectors to routines provided by a symbol reader. These vectors describe routines that operate on the table. The most important routine is the one that converts a partial symbol table to a complete symbol table (psymtab to symtab conversion).

Partial symbols are categorized into two types, global and static. These types correspond to the 'C' language notions of global and static variables. Every partial symbol created is placed into a global array. A partial symbol table, therefore, consists of references into the global array indicating a starting and ending partial symbol. All of the symbols in the array between the starting and ending symbol inclusive, represent the partial symbol table. Figure 7 illustrates this notion.

As suggested above, there are two globally defined arrays, one for global and the other for static partial symbols. These arrays contain all of the partial symbols for the object file. Partial symbols are created as each source file is processed. Thus, the partial symbols created for the global and static symbols in the source file are placed into the global arrays contiguously. In other words, all of the global symbols in a source file have corresponding partial symbols created in the array for global partial symbols. Those symbols follow each other in the global array.

Generally speaking, partially defined symbols and partial symbol tables are created during the initial processing of the object file when the debugger is first executed. This is possible because their creation requires minimal processing of the native symbols in the object file resulting in a fast operation. Also, the partial symbol table contains a reference to the symtab for that source file once it is created. Thus, both the psymtab and symtab tables exists for a given source file. This is done to provide quick access to simple queries about symbols.

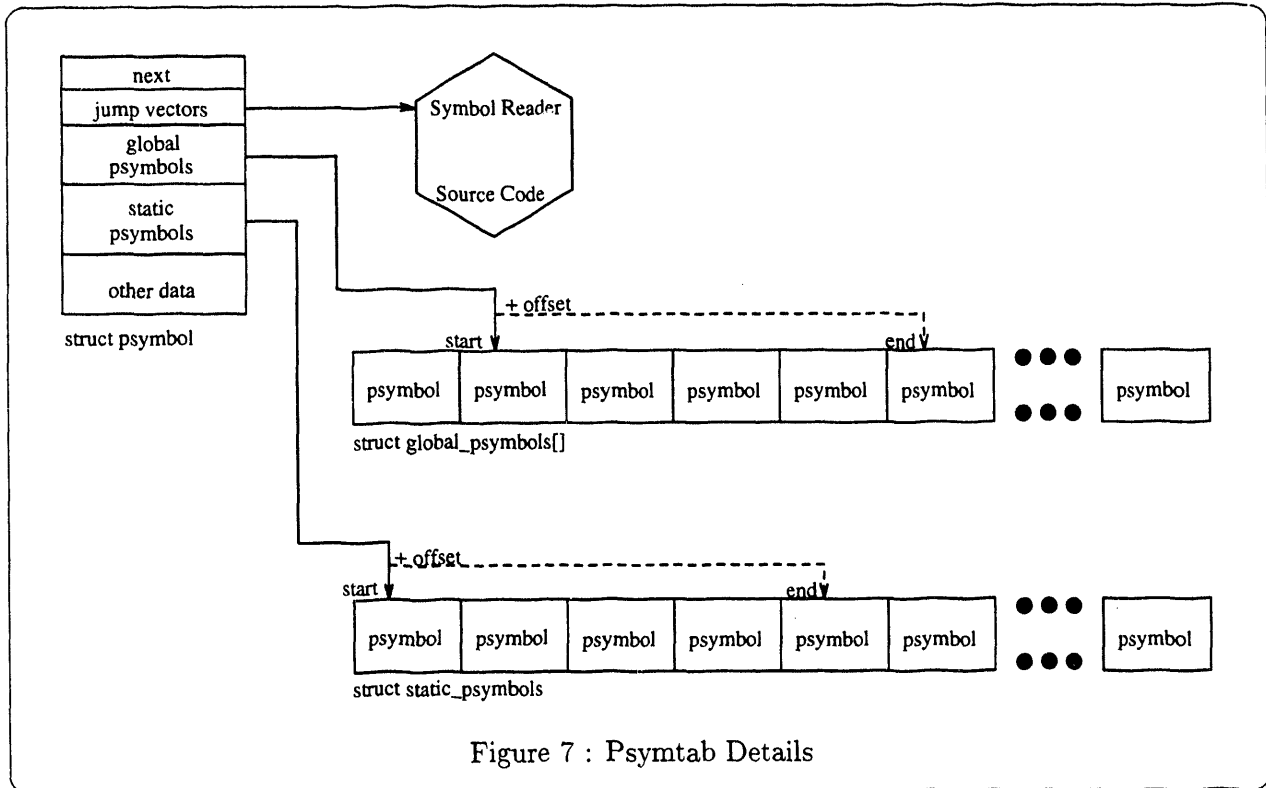


Figure 7 : Psymtab Details

4.3.1.3 Minimal Symbol Table

As its name implies, minimal symbol table provides only a small amount of information about symbols. This table is unlike either the symtab or psymtab tables. The table is not organized by source file. Rather, it contains only global symbols ('C' `extern` symbols) defined in the object file. The table is implemented as a variable sized array consisting of elements of type `struct minimal_symbol`. Each provides a quick mapping of the global symbol to the native address of the symbol in the object file. See Figure 5

4.3.2 Symbols

Corresponding to the three different types of symbol tables are three different types of symbols. The symbols are represented by the structures `struct symbol`, `struct psymbol`, `struct minimal_symbol`. Each structure contains data describing a symbol's name and address. It is the symbol reader's responsibility to generate the symbol structure from the native symbol format in the object file.

5 Remote Debugging Model

The GDB concept of remote debugging provides a way to debug codes on a machine that cannot support an interactive GDB session. Such machines might be an embedded system, or prototype hardware. When debugging with GDB, it defines this type of machine as the *remote*.

The idea behind the remote debugging model is that a host machine running GDB controls an executable on a remote machine. The host communicates with the remote giving it commands to perform, such as viewing registers, or setting breakpoints. The remote then responds to the host, giving the required response to a command, or alerting the host of an event of interest such as a bus error.

5.1 Remote Stub

For the program running on the remote machine to communicate with a GDB host, the remote program must have additional code linked to it. GDB requires that a remote program link with a set of routines, called *stub routines*. These routines provide the necessary functionality to communicate with the host. Also, the routines provide the implementation for certain debugging commands, such as setting a breakpoint, or accessing memory and registers.

The user is responsible for coding and compiling the stub routines on the remote machine. The GDB release provides a few sample stubs for common architectures.

5.2 Communications Model

Communication between the host and the remote can occur in one of two ways. The first is a simple serial line connecting the machines. The second is a TCP/IP connection.[1] It is the responsibility of the stub routines to 'read' from and 'write' to the communication pipeline.

To simplify the communication pipeline between the host and remote, GDB uses a character stream protocol. The stream is divided into packets delimited by the reserved characters $ and #. A complete packet has the following format: *$packet info#checksum*. *Packet info* is a series of characters that represents the data portion of the packet. The data depends on the message

[1] For version 4.9 of GDB, TCP/IP can be used for the stub routines, but there are no details on how to configure the host to use TCP/IP for remote debugging.

being communicated, generally, a debugging command. The debugging commands include: register read/write, memory read/write, resume execution, step execution, and kill.

5.3 Source Files

Figure 8 diagrams the remote debugging model. It also shows how GDB expects to find source file information. Even though the program actually executes on the remote machine, a copy of the executable file and the source files used to build the program are needed on the host machine. The reason for this is to minimize communication between the host and remote. If symbol information about the program can be obtained locally on the host machine, debugging will be quicker. Likewise, viewing the source code will also be faster.[2] Also, if the remote system is robust enough to produce a `core` file, the host can use it during debugging.[3]

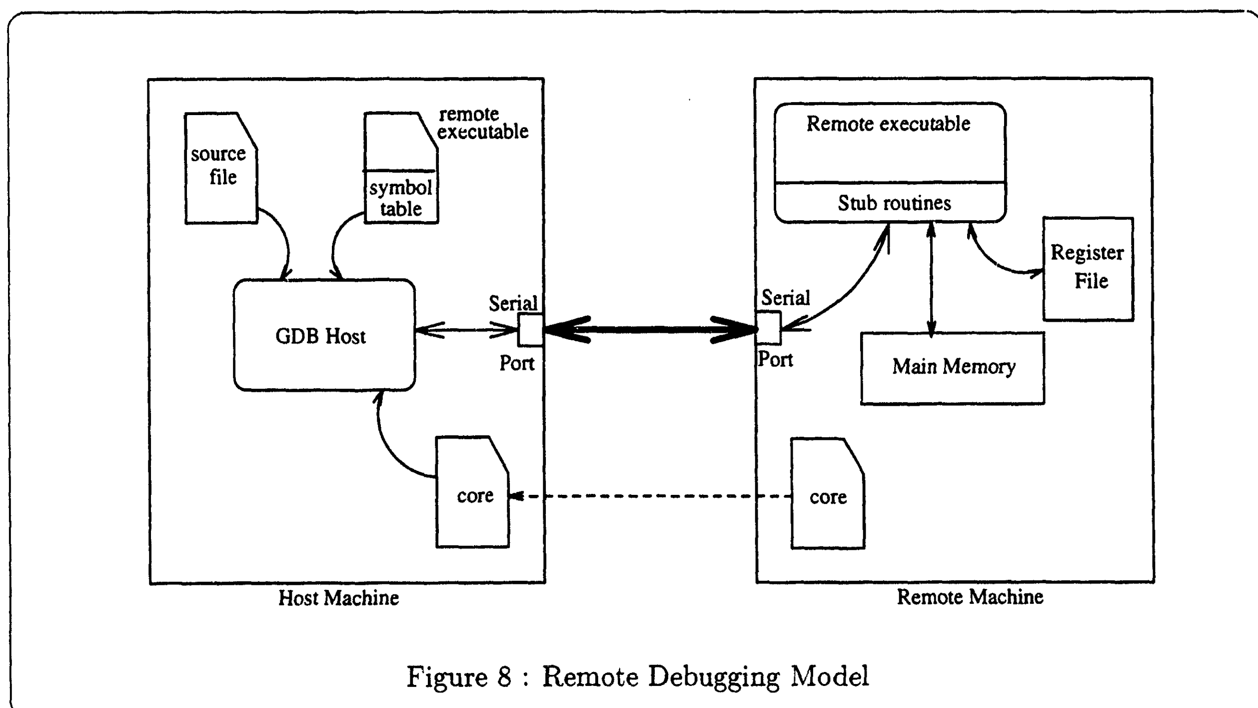


Figure 8 : Remote Debugging Model

[2] This assumes that the GDB host supports the executable format used by the remote machine. If GDB does not support the executable format, then the user must also modify GDB by adding appropriate symbol readers, GDB targets and BFD targets.

[3] Somehow, the `core` file must be copied from the remote to the host.

6 Binary File Descriptor Library

The Binary File Descriptor (BFD) Library is a software tool which provides a common interface to different object file formats.[1] The interface provides routines to access the data within an object file. Access to the object file include reading and writing symbol information, relocation and section information and debugging symbols. As a result of 'understanding' multiple formats, the Library is able to convert data from one format to another.

The BFD Library is split into two parts, the front end, and back end. The front end provides the user[2] with an interface containing canonical data structures, memory management, and routines that act upon object files. The back end deals with the specifics of each type of object file and machine architecture. Each part is detailed below.

6.1 BFD Top Level Architecture

Figure 9 diagrams the top level view of the BFD Library and its interactions between the user and object file. The Library presents the user with *interfaces* between it and the BFD Library, and between the BFD Library and the object file. Through these interfaces, the Library provides a way to decouple the user code from specific object file formats, thus allowing greater portability.

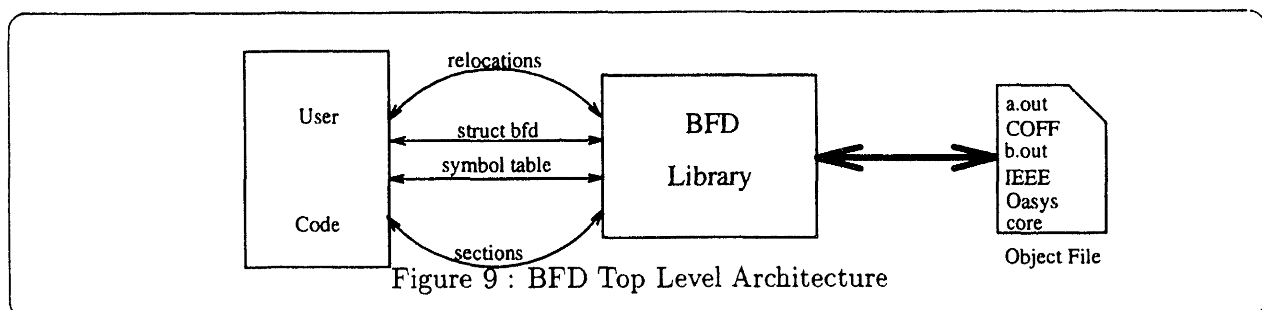


Figure 9 : BFD Top Level Architecture

6.1.1 User - BFD Interface

Elements that are diagrammed represent some of the canonical data structures the Library provides. (See Figure 9). The user calls routines within the Library, passing some or all of these

[1] The BFD Library is a stand-alone tool. It can be used by any program, not just GDB. GDB's usage of the BFD Library is described in Section 4.2.3 [BFD Usage Through Symbol Readers], page 15.

[2] A BFD user is code making BFD library calls.

structures. The desired function required by the user is specified by calling the function named in the Library. The canonical structures used by the Library (as shown in Figure 9) are described below.

relocation A structure containing data that provides a linker with the information necessary to generate the address of a symbol found within the object code. Such information may specify that a symbol has an absolute or indirect reference within a data segment. Normally, the linker uses this information to resolve external declarations of variables, functions, and global data.

BFD structure
The structure provides the basis for almost all routines in the BFD Library. It describes the type of object file, machine architecture, and data used by the BFD back end. See Section 6.3 [BFD Data Structure], page 28.

symbol table
The symbol table structure is an array of pointers to symbols that have been canonized into the corresponding BFD structure. The BFD structure for a symbol is `struct asymbol`. It describes a symbol's name, value, and attributes. The individual symbols are created and placed into the symbol table in the order they exist in the object file.

section A *section* provides an abstraction to maintain and manage the raw data from the object file within a single `struct BFD` instance. Also, some object file formats directly support this abstraction. For example, the '`a.out`' format contains three sections: `.text`, `.data`, `.bss`. The sections represent the code, initialized data, and uninitialized data, respectively.

6.1.2 BFD Library - Object File Interface

The interface to the object file is not directly accessible to the user. It is strictly an interface for the back end portion of the BFD Library. The data between the two is a bidirectional stream of bytes read from and written to the object file. The data is then processed by the back end. See Section 6.2.3 [BFD Back End], page 27.

6.2 BFD Internals

Figure 10 diagrams the internal architecture of the BFD Library. The internals are divided into three parts: front end, library code, back end. Each is described below. In addition, the diagram illustrates more of the canonical structures used to manage the multiple object file formats and machine architectures.

6.2.1 BFD Front End

The front end provides the user with canonical data structures and routines to access the object file. In addition, the front end is responsible for determining the machine architecture and object file format during run-time.

6.2.2 BFD Library Code

This code provides the coupling between the canonical front end and the architecture and machine specific back end. The routines here provide the portability functionality of the BFD Library. This is maintained by jump tables and dynamic arrays used to describe the environment in which the Library runs. Both the front and back ends must call routines in the Library Code, passing only canonical data structures between them.

The Library Code is also responsible for managing and maintaining the necessary structures and jump tables that describe all of the object file formats and machine architectures the Library supports. This is partially done by calling an initialize function for each code module that represents an object file format, or machine architecture.[3] Each initializer is responsible for creating a structure that describes itself in terms of the canonical BFD structures corresponding to what type of code the initializer is for: object file format or machine architecture.

One of the canonical structures used by the Library Code routines is a description of a specific object file format. This description is called a *target* and is represented by the structure `struct bfd_target`. For each target the BFD Library supports, a `struct bfd_target` instance is allocated and placed into a dynamic array of other targets. Figure 10 describes the information the structure contains.[4] Such information includes name of format, general flavor of the format, byte order, etc.

The most critical part of the `struct bfd_target` structure is its jump vectors. The jump vectors (jump table) provides entry points into code that performs specific tasks. The tasks are canonical functions provided to the user through the front end and routines specific to the Library Code.

---

[3] Implementation detail: a shell script is used during the build of the Library that generates the code to call all of the initializers.

[4] Description is the field names of the structure found in `targets.c`

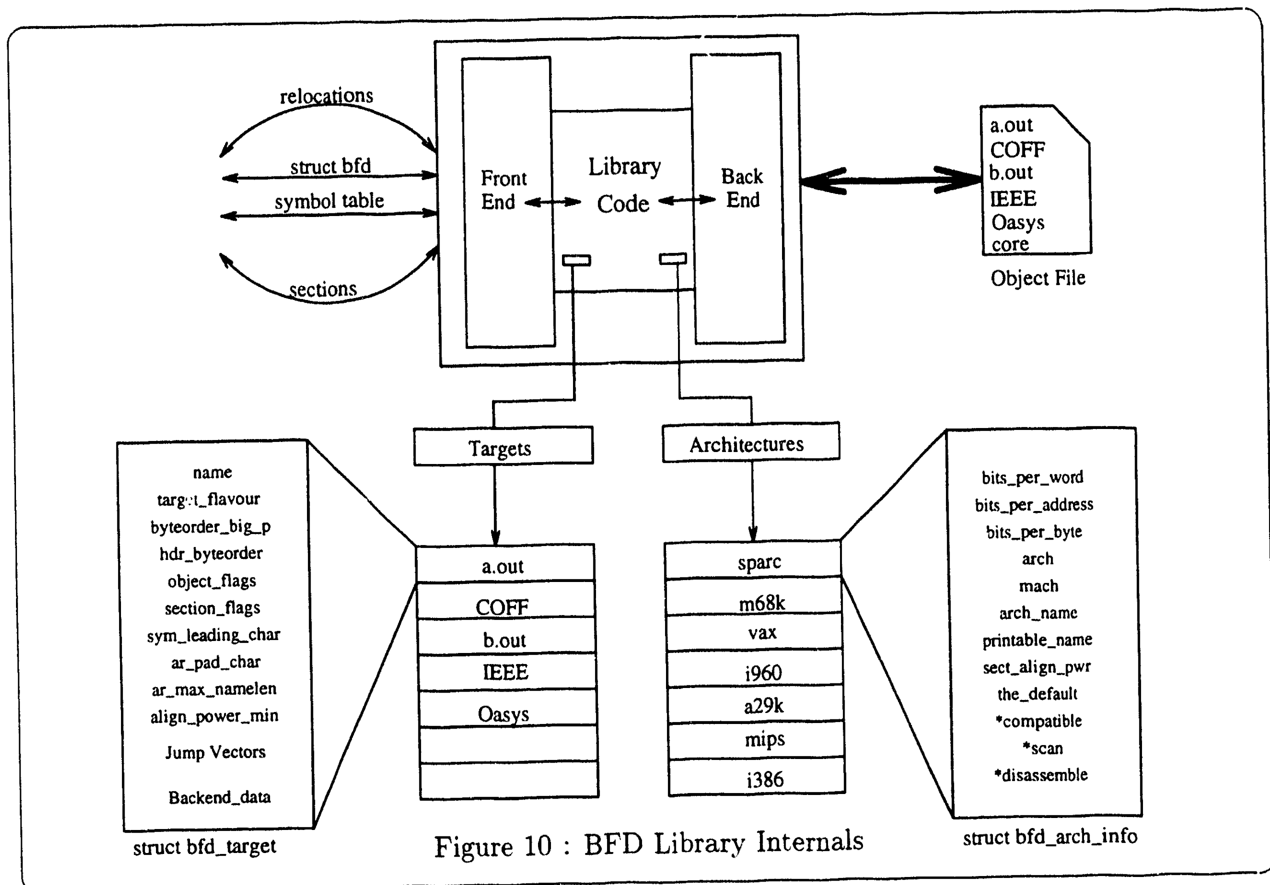


Figure 10 : BFD Library Internals

The other canonical data structure is named **`bfd_arch_info`**, and it manages the data for a machine architecture. This structure is similar to the **`bfd_target`** in the way it is created and managed by the BFD Library. In addition to static data fields, the structure contains function pointers to routines that: determine if an object file is compatible with the architecture, provide disassembly of object code, and scan of an object file for byte patterns.

When the BFD Library is first 'opened' with an object file, all of the architecture structure's compatibility routines are called in sequence. These routines determine if the object file has been linked for the architecture they represent. From the list of architectures, only one routine will recognize the object file. If no routine recognizes the object file format, the BFD Library cannot process it. The other routines found in the **`bfd_arch_info`** structure provides utility routines that access data in the object file.

6.2.3 BFD Back End

The back end deals with the object file directly. Actually, there is not just one back end. The previous sections describe the BFD Library's ability to support multiple object file formats. Thus, it is logical that a separate back end code module is provided for each object file format. Each back

end module is responsible for providing all of the canonical functionality required by the front end and Library Code routines.

The purpose of the `bfd_target` should now be clear. Each of the back end modules for an object file format is managed through the corresponding `bfd_target` for that format. The routines in the back end are thus accessed through the function pointers (jump table, jump vectors) provided in that structure. Thus, from the user's perspective, there is only one back end, hence, portability and extensibility are maximized.

6.3 BFD Data Structure

The BFD structure, `struct _bfd`, is used in almost all calls to the Library. It contains the information the BFD Library needs to fill the user's request. For every object file the user accesses, a unique instance of a BFD structure is created. The important data fields in the structure are diagrammed below. See Figure 11. These include the object file's target, machine architecture, and data storage.

6.3.1 Object File Data

Each object file is described with an instance of a `struct _bfd` instance. Data about the object file's format is a pointer to an instance of a `bfd_target` structure. Through this structure, the routines required by the BFD Library are accessible through the `bfd_target`'s function pointers (jump table, jump vector) to the back end code. For more information about the `bfd_target`, see Section 6.2.2 [BFD Library Code], page 26.

The next element in `bfd_target` is the machine architecture on which the object file executes. This is provided by a pointer to an instance of a `struct bfd_arch_info`. The data in this structure provides the library with information needed to interpret the data in the object file.

6.3.2 Private Store

The BFD Library provides single arrays for both the `bfd_target` and `bfd_arch_info` structures. See Figure 10. Since, multiple `struct _bfd` instances may contain pointers to the same instances of the `bfd_target` and `bfd_arch_info` structures, any action by the Library must use unique memory to store data obtained from the object files represented by the separate `struct _bfd` instances. In other words, the back end code routines must be reentrant. This is accomplished by providing

private static data storage for the back end *in* the **struct _bfd** instance. Thus, a single back end can interact with multiple object files of the same format without corrupting the data.

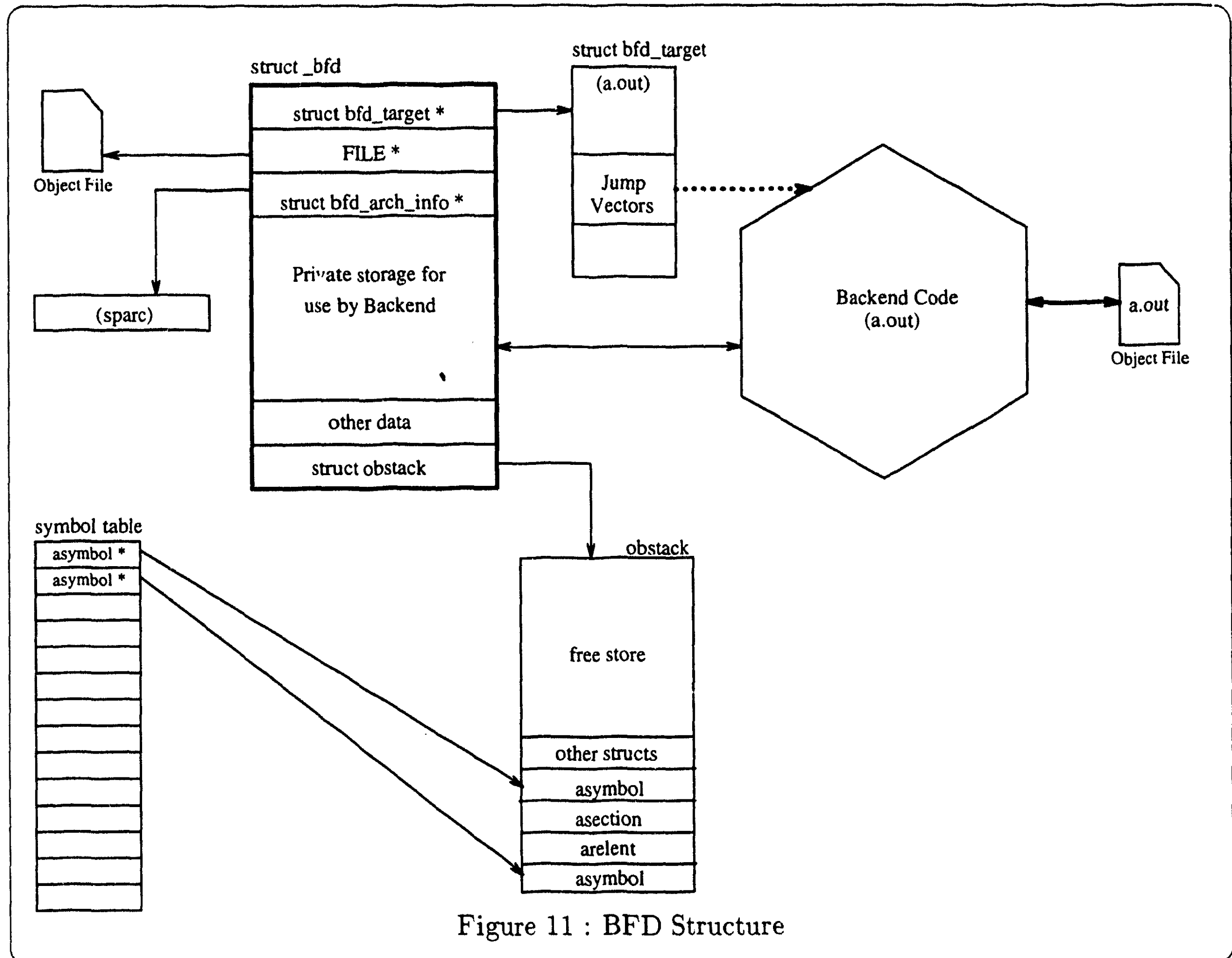


Figure 11 : BFD Structure

6.4 Obstack

The obstack (object stack) provides each BFD structure with its own "heap" from which to allocate dynamic data structures. See Figure 11 for its usage by the BFD Library. When processing the object file, the back end creates instances for the symbols, sections and relocations by using the memory provided in the obstack.

The obstack is a large contiguous block of memory that acts as a 'mini-heap'. Data objects are allocated off of the top of the obstack. The data object may be of undetermined length. This is possible because once an object is created, it may change size indescriminately until another object

is created. At that point, the first object becomes fixed in size. Once the data object's size is fixed, a valid pointer to the beginning of the data is generated. The next data object is created following the last one. The obstack may grow or shrink to accommodate differently sized data objects. Once the data object is created, it is not moved within the object stack.[5]

[5] For more details on the obstack, see the source file `obstack.[ch]`

7 Summary

This report has summarized the major component parts of the GNU debugger. First, the command architecture was described to explain how commands entered from the user are executed. The next chapter began to detail the inferior process GDB uses to control the debugged program. The following chapter then described the symbol tables the debugger uses to manage the object files being debugged. Then, a short discussion about the GDB remote debugging model was presented. Finally, the BFD Library was described. These components form the bulk of the debugger design.

The underlying principle guiding all of the designs within GDB is that of portability. This report has described the structures necessary to maintain that portability. As a result, the GDB debugger is very extensible. If one plans to make modifications, portability must be the primary design consideration. With the structures already in place within the debugger, this is not an impossible constraint. Rather, it is made easier by the canconical structures. As a result, GDB is available on all major platforms and thus is the *de facto* standard for portable debuggers.

Appendix A Source File Summary

This appendix summarizes most of the source files within the GDB release. The files that are not listed explicitly are those that are generated by the configuration shell script during installation of GDB, or are files added since release 4.9 of the debugger. Within this appendix, standard UNIX wildcard filename specifications are used to save space. Fortunately, the filenames of GDB source codes are very descriptive. However, one may still need to examine a source file directly to determinate its exact functionality.

A.1 Source File Summary

Symbol Readers

```
coffread.c          dbxread.c          dwarfread.c    elfread.c
mipsread.c          paread.c           xcoffread.c
```

Inferior Process

```
fork-child.c        infcmd.c           inferior.h     inflow.c
infptrace.c         infrun.c           inftarg.c
```

Symbol Table

```
buildsym.[ch]       objfiles.[ch]      symfile.[ch]   symmisc.[ch]
symtab.[ch]
```

Support Routines

```
alloca.c            c-exp.tab.c        c-exp.y        c-lang.[ch]
c-typeprint.c       c-valprint.c       ch-exp.tab.c   ch-exp.y
ch-lang.[ch]        ch-typeprint.c     ch-valprint.c  cp-cvalprint.c
environ.[ch]        m2-exp.tab.c       m2-exp.y       m2-lang.[ch]
m2-typeprint.[ch]   m2-valprint.[ch]   putenv.[ch]    regex.[ch]
utils.c             valarith.c         valops.c       valprint.c
value.h             values.c           version.c
```

"Core" Routines

```
blockframe.c        breakpoint.[ch]    command.[ch]   complaints.[ch]
core-svr4.c         core.c             coredep.c      corelow.c
```

`defs.h`	`demangle.c`	`eval.c`	`exec.c`
`expprint.c`	`expression.h`	`findvar.c`	`frame.h`
`gdbcmd.h`	`gdbcore.h`	`gdbtypes.[ch]`	`init.c`
`language.[ch]`	`main.c`	`maint.c`	`mem-break.c`
`minimon.h`	`minisyms.c`	`parse.c`	`parser-defs.h`
`partial-stab.h`	`printcmd.c`	`solib.[ch]`	`source.c`
`stabsread.[ch]`	`stack.c`	`standalone.c`	`stuff.c`
`typeprint.[ch]`			

A.2 Remote Debugging Routines

The source files that handle remote debugging are all specified by the following convention: `remote-`*machineID*`.[ch]`, where *machineID* is a description of the machine architecture name for the source file. For example, the file `remote-mips.c` contains the remote debugging routines for a MIPS architecture. Also, the files: `remote.c, ser-*.c, serial.[ch]` are used with remote debugging.

A.3 Target Dependent Codes

The source files that handle machine dependent code are all specified by the following convention: *machineID-sourceType*`.[ch]`, where *machineID* is a description of the machine architecture and *sourceType* is a description of the type of dependent code the file contains. The types of dependent code are as follows:

`*-nat.c` native dependent code

`*-pinst.c`
print machine-level instruction opcodes

`*-tdep.c` code for GDB target structure dependent on architecture

`*-xdep.c` debugging machine host dependent code

A.4 GDB Source Tree

The complete GDB source directory tree contains a significant number of subdirectories, of which only one contains the source file for the debugger, as described above. Fortunately, all of the directory names are descriptive. In addition to the filename formats described above, the following formats are also defined:

`*.mh`	Makefile configuration parameters
`*.mt`	Makefile configuration parameters for target dependent files
`xm-*.h`	Global `#includes` and `#defines` for host dependent files
`*-xdep.c`	Global variables and routines for host dependent files
`nm-*.h`	Extra `#includes` and `#defines` for native dependent files
`*-nat.c`	Global variables and routines for native dependent files
`tm-*.h`	Global `#includes` and `#defines` for target dependent files
`*-tdep.c`	Global variables and routines for target dependent files

Appendix B Call Graphs

This appendix contains a few call graphs derived from gdb source code. These are included here just as a sample. The graphs are just too large to squeeze onto one page, so I have eliminated calls to standard routines (free, malloc, etc.).

In addition to these sample graphs, I have created complete graphs for the following code modules. See Section A.1 [Source File Summary], page 32 for contents of the modules.

- Symbol Reader (`coffread.c`)
- Inferior Process
- Symbol Table
- "Core" Routine (`main.c`)

The complete call graphs are not supplied with this document. You can obtain the graphs from the same ftp site that you can obtain this report. See Chapter 1 [Introduction], page 1, for site information. The graphs are large postscript files that create a "mosaic" of the graph. In other words, it's broken into many segments printed on multiple pages.

Fortunately, the graphs are generated automatically using a program from AT&T called `dot`. More information about this graphing tool may be obtained via anonymous ftp from the site `research.att.com`. Documents can then be found in the `dist/drawdag` directory. Also, I have a couple of shell scripts and a small program that will take "C" source and create the input that `dot` uses to create a graph. These are also included in the ftp site as mentioned in the Introduction.

B.1 Symtab Graph

The following figure is a partial call graph for all routines in the file: `symtab.c`.

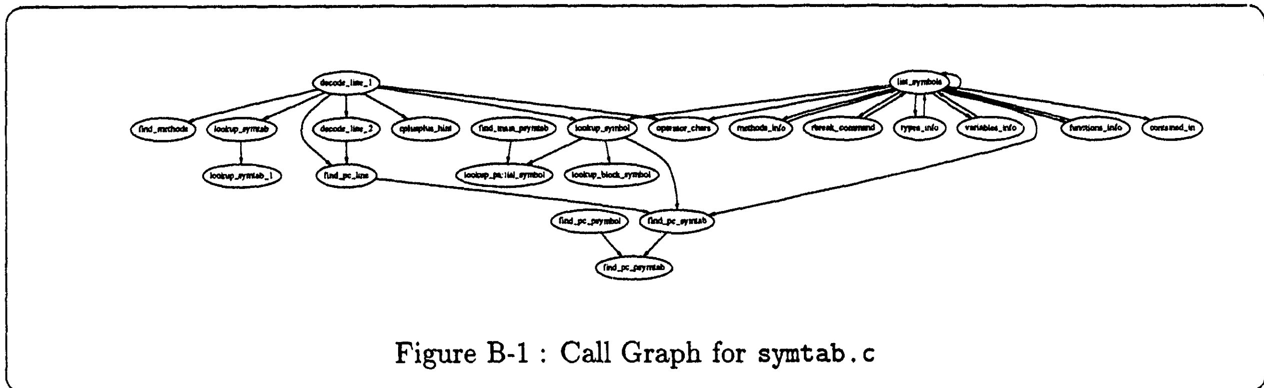


Figure B-1 : Call Graph for `symtab.c`

B.2 Command Graph

The following figure is a partial call graph for all routines in the file: `command.c`.

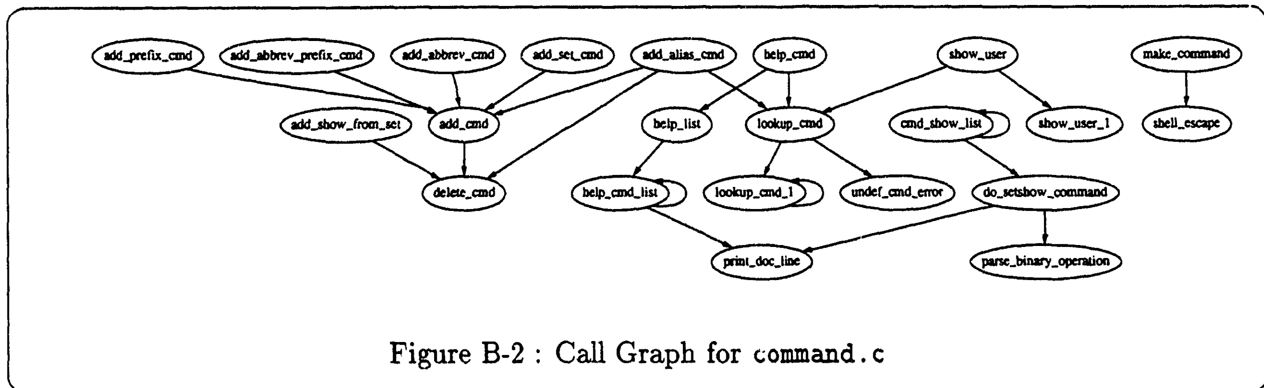


Figure B-2 : Call Graph for `command.c`

DATE
FILMED
3/8/94

END