libbfd

The Binary File Descriptor Library

First Edition—BFD version < 2.0
April 1991

Steve Chamberlain
Cygnus Support

Cygnus Support
sac@cygnus.com
BFD, Revision: 1.12
TEXinfo 2.60

Copyright © 1991 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, subject to the terms of the GNU General Public License, which includes the provision that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

1 Introduction

Simply put, BFD is a package which allows applications to use the same routines to operate on object files whatever the object file format. A different object file format can be supported simply by creating a new BFD back end and adding it to the library.

BFD is split into two parts; the front end and the many back ends.

- The front end of BFD provides the interface to the user. It manages memory, and various canonical data structures. The front end also decides which back end to use, and when to call back end routines.
- The back ends provide BFD its view of the real world. Each back end provides a set of calls which the BFD front end can use to maintain its canonical form. The back ends also may keep around information for their own use, for greater efficiency.

1.1 History

One spur behind BFD was the desire, on the part of the GNU 960 team at Intel Oregon, for interoperability of applications on their COFF and b.out file formats. Cygnus was providing GNU support for the team, and Cygnus was contracted to provide the required functionality.

The name came from a conversation David Wallace was having with Richard Stallman about the library: RMS said that it would be quite hard—David said "BFD". Stallman was right, but the name stuck.

At the same time, Ready Systems wanted much the same thing, but for different object file formats: IEEE-695, Oasys, Srecords, a.out and 68k coff.

BFD was first implemented by members of Cygnus Support; Steve Chamberlain (sac@cygnus.com), John Gilmore (gnu@cygnus.com), K. Richard Pixley (rich@cygnus.com) and David Henkel-Wallace (gumby@cygnus.com).

1.2 How It Works

To use the library, include `bfd.h` and link with `libbfd.a`.

BFD provides a common interface to the parts of an object file for a calling application.

When an application sucessfully opens a target file (object, archive or whatever) a pointer to an internal structure is returned. This pointer points to a structure called `bfd`, described in `include/bfd.h`. Our convention is to call this pointer a BFD, and instances of it within code `abfd`. All operations on the target object file are applied as methods to the BFD. The mapping is defined within `bfd.h` in a set of macros, all beginning '`bfd`'_.

For example, this sequence would do what you would probably expect: return the number of sections in an object file attached to a BFD `abfd`.

```
#include "bfd.h"

unsigned int number_of_sections(abfd)
bfd *abfd;
{
  return bfd_count_sections(abfd);
}
```

The abstraction used within BFD is that an object file has a header, a number of sections containing raw data, a set of relocations, and some symbol information. Also, BFDs opened for archives have the additional attribute of an index and contain subordinate BFDs. This approach is fine for a.out and coff, but loses efficiency when applied to formats such as S-records and IEEE-695.

1.3 What BFD Version 1 Can Do

As different information from the the object files is required, BFD reads from different sections of the file and processes them. For example a very common operation for the linker is processing symbol tables. Each BFD back end provides a routine for converting between the object file's representation of symbols and an internal canonical format. When the linker asks for the symbol table of an object file, it calls through the memory pointer to the relevant BFD back end routine which reads and converts the table into a canonical form. The linker then operates upon the canonical form. When the link is finished and the linker writes the output file's symbol table, another BFD back end routine is called which takes the newly created symbol table and converts it into the chosen output format.

1.3.1 Information Loss

Some information is lost due to the nature of the file format. The output targets supported by BFD do not provide identical facilities, and information which may be described in one form has nowhere to go in another format. One example of this is alignment information in `b.out`. There is nowhere in an `a.out` format file to store alignment information on the contained data, so when a file is linked from `b.out` and an `a.out` image is produced, alignment information will not propagate to the output file. (The linker will still use the alignment information internally, so the link is performed correctly).

Another example is COFF section names. COFF files may contain an unlimited number of sections, each one with a textual section name. If the target of the link is a format which does not have many sections (eg `a.out`) or has sections without names (eg the Oasys format) the link cannot be done simply. You can circumvent this problem by describing the desired input-to-output section mapping with the linker command language.

Information can be lost during canonicalization. The BFD internal canonical form of the external formats is not exhaustive; there are structures in input formats for which there is no direct representation internally. This means that the BFD back ends cannot maintain all possible data richness through the transformation between external to internal and back to external formats.

This limitation is only a problem when an application reads one format and writes another. Each BFD back end is responsible for maintaining as much data as possible, and the internal BFD canonical form has structures which are opaque to the BFD core, and exported only to the back ends. When a file is read in one format, the canonical form is generated for BFD and the application. At the same time, the back end saves away any information which may otherwise be lost. If the data is then written back in the same format, the back end routine will be able to use the canonical form provided by the BFD core as well as the information it prepared earlier. Since there is a great deal of commonality between back ends, this mechanism is very useful. There is no information lost for this reason when linking or copying big endian COFF to little endian COFF, or `a.out` to `b.out`. When a mixture of formats is linked, the information is only lost from the files whose format differs from the destination.

1.3.2 Mechanism

The greatest potential for loss of information is when there is least overlap between the information provided by the source format, that stored by the canonical format, and the information needed by the destination format. A brief description of the canonical form may help you appreciate what

kinds of data you can count on preserving across conversions.

files Information on target machine architecture, particular implementation and format type are stored on a per-file basis. Other information includes a demand pageable bit and a write protected bit. Note that information like Unix magic numbers is not stored here—only the magic numbers' meaning, so a `ZMAGIC` file would have both the demand pageable bit and the write protected text bit set. The byte order of the target is stored on a per-file basis, so that big- and little-endian object files may be linked with one another.

sections Each section in the input file contains the name of the section, the original address in the object file, various flags, size and alignment information and pointers into other BFD data structures.

symbols Each symbol contains a pointer to the object file which originally defined it, its name, its value, and various flag bits. When a BFD back end reads in a symbol table, the back end relocates all symbols to make them relative to the base of the section where they were defined. This ensures that each symbol points to its containing section. Each symbol also has a varying amount of hidden data to contain private data for the BFD back end. Since the symbol points to the original file, the private data format for that symbol is accessible. `gld` can operate on a collection of symbols of wildly different formats without problems.

Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out` type information is stored in the symbol table as long symbol names. This information would be useless to most COFF debuggers; the linker has command line switches to allow users to throw it away.

There is one word of type information within the symbol, so if the format supports symbol type information within symbols (for example COFF, IEEE, Oasys) and the type is simple enough to fit within one word (nearly everything but aggregates) the information will be preserved.

relocation level

Each canonical BFD relocation record contains a pointer to the symbol to relocate to, the offset of the data to relocate, the section the data is in and a pointer to a relocation type descriptor. Relocation is performed effectively by message passing through the relocation type descriptor and symbol pointer. It allows relocations to be performed on output data using a relocation method only available in one of the input formats. For instance, Oasys provides a byte relocation format. A relocation record requesting this relocation type would point indirectly to a routine to perform this, so the relocation may be performed on a byte being written to a COFF file, even though 68k COFF has

no such relocation type.

line numbers

Object formats can contain, for debugging purposes, some form of mapping between symbols, source line numbers, and addresses in the output file. These addresses have to be relocated along with the symbol information. Each symbol with an associated list of line number records points to the first record of the list. The head of a line number list consists of a pointer to the symbol, which allows divination of the address of the function whose line number is being described. The rest of the list is made up of pairs: offsets into the section and line numbers. Any format which can simply derive this information can pass it successfully between formats (COFF, IEEE and Oasys).

2 BFD front end

2.1 typedef bfd

A BFD is has type `bfd`; objects of this type are the cornerstone of any application using `libbfd`. References though the BFD and to data in the BFD give the entire BFD functionality. Here is the struct used to define the type `bfd`. This contains he major data about the file, and contains pointers to the rest of the data.

.

```
struct _bfd
{
     /* The filename the application opened the BFD with.  */
    CONST char *filename;

     /* A pointer to the target jump table.             */
    struct bfd_target *xvec;

     /* To avoid dragging too many header files into every file that
       includes `<<bfd.h>>', IOSTREAM has been declared as a "char
       *", and MTIME as a "long".  Their correct types, to which they
       are cast when used, are "FILE *" and "time_t".    The iostream
       is the result of an fopen on the filename. */
    char *iostream;

     /* Is the file being cached */

    boolean cacheable;

     /* Marks whether there was a default target specified when the
       BFD was opened. This is used to select what matching algorithm
       to use to chose the back end. */

    boolean target_defaulted;

     /* The caching routines use these to maintain a
       least-recently-used list of BFDs */

    struct _bfd *lru_prev, *lru_next;

     /* When a file is closed by the caching routines, BFD retains
       state information on the file here:
     */

    file_ptr where;
```

```
 /* and here:*/

boolean opened_once;

 /* Set if we have a locally maintained mtime value, rather than
   getting it from the file each time: */

boolean mtime_set;

 /* File modified time, if mtime_set is true: */

long mtime;

 /* Reserved for an unimplemented file locking extension.*/

int ifd;

 /* The format which belongs to the BFD.*/

bfd_format format;

 /* The direction the BFD was opened with*/

enum bfd_direction {no_direction = 0,
                    read_direction = 1,
                    write_direction = 2,
                    both_direction = 3} direction;

 /* Format_specific flags*/

flagword flags;

 /* Currently my_archive is tested before adding origin to
   anything. I believe that this can become always an add of
   origin, with origin set to 0 for non archive files.   */

file_ptr origin;

 /* Remember when output has begun, to stop strange things
   happening. */
boolean output_has_begun;

 /* Pointer to linked list of sections*/
struct sec  *sections;

 /* The number of sections */
unsigned int section_count;

 /* Stuff only useful for object files:
   The start address. */
```

```
    bfd_vma start_address;

     /* Used for input and output*/
    unsigned int symcount;

     /* Symbol table for output BFD*/
    struct symbol_cache_entry  **outsymbols;

     /* Pointer to structure which contains architecture information*/
    struct bfd_arch_info *arch_info;

     /* Stuff only useful for archives:*/
    PTR arelt_data;
    struct _bfd *my_archive;
    struct _bfd *next;
    struct _bfd *archive_head;
    boolean has_armap;

     /* Used by the back end to hold private data. */

    union
      {
      struct aout_data_struct *aout_data;
      struct artdata *aout_ar_data;
      struct _oasys_data *oasys_obj_data;
      struct _oasys_ar_data *oasys_ar_data;
      struct coff_tdata *coff_obj_data;
      struct ieee_data_struct *ieee_data;
      struct ieee_ar_data_struct *ieee_ar_data;
      struct srec_data_struct *srec_data;
      struct elf_obj_tdata_struct *elf_obj_data;
      struct elf_core_tdata_struct *elf_core_data;
      struct bout_data_struct *bout_data;
      struct sun_core_struct *sun_core_data;
      struct trad_core_struct *trad_core_data;
      PTR any;
      } tdata;

     /* Used by the application to hold private data*/
    PTR usrdata;

     /* Where all the allocated stuff under this BFD goes */
    struct obstack memory;

    asymbol **ld_symbols;
};
```

2.1.0.1 bfd_get_reloc_upper_bound

Synopsis

```
unsigned int bfd_get_reloc_upper_bound(bfd *abfd, asection *sect);
```

Description

This function return the number of bytes required to store the relocation information associated with section `sect` attached to bfd `abfd`

2.1.0.2 bfd_canonicalize_reloc

Synopsis

```
unsigned int bfd_canonicalize_reloc
   (bfd *abfd,
    asection *sec,
    arelent **loc,
    asymbol **syms);
```

Description

This function calls the back end associated with the open `abfd` and translates the external form of the relocation information attached to `sec` into the internal canonical form. The table is placed into memory at `loc`, which has been preallocated, usually by a call to `bfd_get_reloc_upper_bound`. The `syms` table is also needed for horrible internal magic reasons.

2.1.0.3 bfd_set_file_flags

Synopsis

```
boolean bfd_set_file_flags(bfd *abfd, flagword flags);
```

Description

This function attempts to set the flag word in the referenced BFD structure to the value supplied. Possible errors are:

- wrong_format - The target bfd was not of object format.
- invalid_operation - The target bfd was open for reading.
- invalid_operation - The flag word contained a bit which was not applicable to the type of file. eg, an attempt was made to set the D_PAGED bit on a bfd format which does not support demand paging

2.1.0.4 bfd_set_reloc

Synopsis

```
void bfd_set_reloc
   (bfd *abfd, asection *sec, arelent **rel, unsigned int count)
```

Description

This function sets the relocation pointer and count within a section to the supplied values.

2.1.0.5 bfd_set_start_address

Description

Marks the entry point of an output BFD.

Returns

Returns `true` on success, `false` otherwise.

Synopsis

```
boolean bfd_set_start_address(bfd *, bfd_vma);
```

2.1.0.6 The bfd_get_mtime function

Synopsis

```
long bfd_get_mtime(bfd *);
```

Description

Return file modification time (as read from file system, or from archive header for archive members).

2.1.0.7 stuff

Description

stuff which should be documented

```
#define bfd_sizeof_headers(abfd, reloc) \
     BFD_SEND (abfd, _bfd_sizeof_headers, (abfd, reloc))
```

```
#define bfd_find_nearest_line(abfd, sec, syms, off, file, func, line) \
     BFD_SEND (abfd, _bfd_find_nearest_line,  (abfd, sec, syms, off, file, func,
line))

#define bfd_debug_info_start(abfd) \
        BFD_SEND (abfd, _bfd_debug_info_start, (abfd))

#define bfd_debug_info_end(abfd) \
        BFD_SEND (abfd, _bfd_debug_info_end, (abfd))

#define bfd_debug_info_accumulate(abfd, section) \
        BFD_SEND (abfd, _bfd_debug_info_accumulate, (abfd, section))

#define bfd_stat_arch_elt(abfd, stat) \
        BFD_SEND (abfd, _bfd_stat_arch_elt,(abfd, stat))

#define bfd_coff_swap_aux_in(a,e,t,c,i) \
        BFD_SEND (a, _bfd_coff_swap_aux_in, (a,e,t,c,i))

#define bfd_coff_swap_sym_in(a,e,i) \
        BFD_SEND (a, _bfd_coff_swap_sym_in, (a,e,i))

#define bfd_coff_swap_lineno_in(a,e,i) \
        BFD_SEND ( a, _bfd_coff_swap_lineno_in, (a,e,i))

#define bfd_set_arch_mach(abfd, arch, mach)\
        BFD_SEND ( abfd, _bfd_set_arch_mach, (abfd, arch, mach))

#define bfd_coff_swap_reloc_out(abfd, i, o) \
        BFD_SEND (abfd, _bfd_coff_swap_reloc_out, (abfd, i, o))

#define bfd_coff_swap_lineno_out(abfd, i, o) \
        BFD_SEND (abfd, _bfd_coff_swap_lineno_out, (abfd, i, o))

#define bfd_coff_swap_aux_out(abfd, i, t,c,o) \
        BFD_SEND (abfd, _bfd_coff_swap_aux_out, (abfd, i,t,c, o))

#define bfd_coff_swap_sym_out(abfd, i,o) \
        BFD_SEND (abfd, _bfd_coff_swap_sym_out, (abfd, i, o))

#define bfd_coff_swap_scnhdr_out(abfd, i,o) \
        BFD_SEND (abfd, _bfd_coff_swap_scnhdr_out, (abfd, i, o))

#define bfd_coff_swap_filehdr_out(abfd, i,o) \
        BFD_SEND (abfd, _bfd_coff_swap_filehdr_out, (abfd, i, o))

#define bfd_coff_swap_aouthdr_out(abfd, i,o) \
        BFD_SEND (abfd, _bfd_coff_swap_aouthdr_out, (abfd, i, o))
```

```
#define bfd_get_relocated_section_contents(abfd, seclet) \
BFD_SEND (abfd, _bfd_get_relocated_section_contents, (abfd, seclet))

#define bfd_relax_section(abfd, section, symbols) \
       BFD_SEND (abfd, _bfd_relax_section, (abfd, section, symbols))
```

2.2 Memory Usage

BFD keeps all its internal structures in obstacks. There is one obstack per open BFD file, into which the current state is stored. When a BFD is closed, the obstack is deleted, and so everything which has been allocated by libbfd for the closing file will be thrown away.

BFD will not free anything created by an application, but pointers into `bfd` structures will be invalidated on a `bfd_close`; for example, after a `bfd_close` the vector passed to `bfd_canonicalize_symtab` will still be around, since it has been allocated by the application, but the data that it pointed to will be lost.

The general rule is not to close a BFD until all operations dependent upon data from the BFD have been completed, or all the data from within the file has been copied. To help with the management of memory, there is a function (`bfd_alloc_size`) which returns the number of bytes in obstacks associated with the supplied BFD. This could be used to select the greediest open BFD, close it to reclaim the memory, perform some operation and reopen the BFD again, to get a fresh copy of the data structures.

2.3 Initialization

This is the initialization section

2.3.0.1 `bfd_init`

Synopsis

```
void bfd_init(void);
```

Description
This routine must be called before any other bfd function to initialize magical internal data structures.

2.3.0.2 bfd_check_init

Description
This routine is called before any other bfd function using initialized data is used to ensure that the structures have been initialized. Soon this function will go away, and the bfd library will assume that bfd_init has been called.
Synopsis

```
void bfd_check_init(void);
```

2.4 Sections

Sections are supported in BFD in `section.c`. The raw data contained within a BFD is maintained through the section abstraction. A single BFD may have any number of sections, and keeps hold of them by pointing to the first, each one points to the next in the list.

2.4.1 Section Input

When a BFD is opened for reading, the section structures are created and attached to the BFD. Each section has a name which describes the section in the outside world - for example, `a.out` would contain at least three sections, called `.text`, `.data` and `.bss`. Sometimes a BFD will contain more than the 'natural' number of sections. A back end may attach other sections containing constructor data, or an application may add a section (using bfd_make_section) to the sections attached to an already open BFD. For example, the linker creates a supernumary section `COMMON` for each input file's BFD to hold information about common storage. The raw data is not necessarily read in at the same time as the section descriptor is created. Some targets may leave the data in place until a `bfd_get_section_contents` call is made. Other back ends may read in all the data at once - For example; an S-record file has to be read once to determine the size of the data. An IEEE-695 file doesn't contain raw data in sections, but data and relocation expressions intermixed, so the data area has to be parsed to get out the data and relocations.

2.4.2 Section Output

To write a new object style BFD, the various sections to be written have to be created. They are attached to the BFD in the same way as input sections, data is written to the sections using `bfd_set_section_contents`. The linker uses the fields `output_section` and `output_offset` to create an output file. The data to be written comes from input sections attached to the output sections. The output section structure can be considered a filter for the input section, the output section determines the vma of the output data and the name, but the input section determines the offset into the output section of the data to be written. Eg to create a section "O", starting at 0x100, 0x123 long, containing two subsections, "A" at offset 0x0 (ie at vma 0x100) and "B" at offset 0x20 (ie at vma 0x120) the structures would look like:

```
section name          "A"
  output_offset   0x00
  size            0x20
  output_section ----------->  section name    "O"
                          |    vma             0x100
section name          "B" |    size            0x123
  output_offset   0x20    |
  size            0x103   |
  output_section  --------|
```

2.4.3 Seglets

The data within a section is stored in a `seglet`. These are much like the fixups in `gas`. The seglet abstraction allows the a section to grow and shrink within itself. A seglet knows how big it is, and which is the next seglet and where the raw data for it is, and also points to a list of relocations which apply to it. The seglet is used by the linker to perform relaxing on final code. The application creates code which is as big as necessary to make it work without relaxing, and the user can select whether to relax. Sometimes relaxing takes a lot of time. The linker runs around the relocations to see if any are attached to data which can be shrunk, if so it does it on a seglet by seglet basis.

2.4.4 typedef asection

The shape of a section struct:

.

```
typedef struct sec
```

```
{
         /* The name of the section, the name isn't a copy, the pointer is
        the same as that passed to bfd_make_section. */

    CONST char *name;


         /* Which section is it 0.nth      */

   int index;

         /* The next section in the list belonging to the BFD, or NULL. */

    struct sec *next;

         /* The field flags contains attributes of the section. Some of
           flags are read in from the object file, and some are
           synthesized from other information.  */

    flagword flags;

#define SEC_NO_FLAGS   0x000

         /* Tells the OS to allocate space for this section when loaded.
           This would clear for a section containing debug information
           only. */


#define SEC_ALLOC      0x001
         /* Tells the OS to load the section from the file when loading.
           This would be clear for a .bss section */

#define SEC_LOAD       0x002
         /* The section contains data still to be relocated, so there will
           be some relocation information too. */

#define SEC_RELOC      0x004

         /* Obsolete ? */

#define SEC_BALIGN     0x008

         /* A signal to the OS that the section contains read only
          data. */
#define SEC_READONLY   0x010

         /* The section contains code only. */

#define SEC_CODE       0x020
```

```
         /* The section contains data only. */

#define SEC_DATA        0x040

         /* The section will reside in ROM. */

#define SEC_ROM        0x080

         /* The section contains constructor information. This section
           type is used by the linker to create lists of constructors and
           destructors used by <<g++>>. When a back end sees a symbol
           which should be used in a constructor list, it creates a new
           section for the type of name (eg <<__CTOR_LIST__>>), attaches
           the symbol to it and builds a relocation. To build the lists
           of constructors, all the linker has to to is catenate all the
           sections called <<__CTOR_LIST__>> and relocte the data
           contained within - exactly the operations it would peform on
           standard data. */

#define SEC_CONSTRUCTOR 0x100

         /* The section is a constuctor, and should be placed at the
            end of the . */


#define SEC_CONSTRUCTOR_TEXT 0x1100

#define SEC_CONSTRUCTOR_DATA 0x2100

#define SEC_CONSTRUCTOR_BSS  0x3100


         /* The section has contents - a bss section could be
           <<SEC_ALLOC>> | <<SEC_HAS_CONTENTS>>, a debug section could be
           <<SEC_HAS_CONTENTS>> */

#define SEC_HAS_CONTENTS 0x200

         /* An instruction to the linker not to output sections
           containing this flag even if they have information which
           would normally be written. */

#define SEC_NEVER_LOAD 0x400



   bfd_vma vma;

         /* The size of the section in bytes, as it will be output.
           contains a value even if the section has no contents (eg, the
```

```
        size of <<.bss>>). This will be filled in after relocation */

bfd_size_type _cooked_size;

      /* The size on disk of the section in bytes originally.  Normally this
 value is the same as the size, but if some relaxing has
 been done, then this value will be bigger.  */

bfd_size_type _raw_size;

      /* If this section is going to be output, then this value is the
        offset into the output section of the first byte in the input
        section. Eg, if this was going to start at the 100th byte in
        the output section, this value would be 100. */

bfd_vma output_offset;

      /* The output section through which to map on output. */

struct sec *output_section;

      /* The alignment requirement of the section, as an exponent - eg
        3 aligns to 2^3 (or 8) */

unsigned int alignment_power;

      /* If an input section, a pointer to a vector of relocation
        records for the data in this section. */

struct reloc_cache_entry *relocation;

      /* If an output section, a pointer to a vector of pointers to
        relocation records for the data in this section. */

struct reloc_cache_entry **orelocation;

      /* The number of relocation records in one of the above  */

unsigned reloc_count;

      /* Information below is back end specific - and not always used
        or updated

        File position of section data    */

file_ptr filepos;

      /* File position of relocation info */

file_ptr rel_filepos;
```

```
      /* File position of line data       */

   file_ptr line_filepos;

      /* Pointer to data for applications */

   PTR userdata;

   struct lang_output_section *otheruserdata;

      /* Attached line number information */

   alent *lineno;

      /* Number of line number records   */

   unsigned int lineno_count;

      /* When a section is being output, this value changes as more
        linenumbers are written out */

   file_ptr moving_line_filepos;

      /* what the section number is in the target world  */

   int target_index;

   PTR used_by_bfd;

      /* If this is a constructor section then here is a list of the
        relocations created to relocate items within it. */

   struct relent_chain *constructor_chain;

      /* The BFD which owns the section. */

   bfd *owner;

   boolean reloc_done;
  /* A symbol which points at this section only */
   struct symbol_cache_entry *symbol;
   struct symbol_cache_entry **symbol_ptr_ptr;
   struct bfd_seclet_struct *seclets_head;
   struct bfd_seclet_struct *seclets_tail;
} asection ;


#define BFD_ABS_SECTION_NAME "*ABS*"
#define BFD_UND_SECTION_NAME "*UND*"
```

```
#define BFD_COM_SECTION_NAME "*COM*"

     /* the absolute section */
 extern   asection bfd_abs_section;
     /* Pointer to the undefined section */
 extern   asection bfd_und_section;
     /* Pointer to the common section */
 extern asection bfd_com_section;

 extern struct symbol_cache_entry *bfd_abs_symbol;
 extern struct symbol_cache_entry *bfd_com_symbol;
 extern struct symbol_cache_entry *bfd_und_symbol;
#define bfd_get_section_size_before_reloc(section) \
     (section->reloc_done ? (abort(),1): (section)->_raw_size)
#define bfd_get_section_size_after_reloc(section) \
     ((section->reloc_done) ? (section)->_cooked_size: (abort(),1))
```

2.4.5 section prototypes

These are the functions exported by the section handling part of `libbfd`

2.4.5.1 bfd_get_section_by_name

Synopsis

```
asection *bfd_get_section_by_name(bfd *abfd, CONST char *name);
```

Description

Runs through the provided *abfd* and returns the `asection` who's name matches that provided, otherwise NULL. See ⟨undefined⟩ [Sections], page ⟨undefined⟩, for more information.

2.4.5.2 bfd_make_section_old_way

Synopsis

```
asection *bfd_make_section_old_way(bfd *, CONST char *name);
```

Description

This function creates a new empty section called *name* and attaches it to the end of the chain of sections for the BFD supplied. An attempt to create a section with a name which is already in use, returns its pointer without changing the section chain. It has the funny name since this is the way it used to be before is was rewritten... Possible errors are:

- invalid_operation If output has already started for this BFD.
- no_memory If obstack alloc fails.

2.4.5.3 bfd_make_section

Synopsis

```
asection * bfd_make_section(bfd *, CONST char *name);
```

Description

This function creates a new empty section called *name* and attaches it to the end of the chain of sections for the BFD supplied. An attempt to create a section with a name which is already in use, returns NULL without changing the section chain. Possible errors are:

- invalid_operation - If output has already started for this BFD.
- no_memory - If obstack alloc fails.

2.4.5.4 bfd_set_section_flags

Synopsis

```
boolean bfd_set_section_flags(bfd *, asection *, flagword);
```

Description

Attempts to set the attributes of the section named in the BFD supplied to the value. Returns true on success, false on error. Possible error returns are:

- invalid operation The section cannot have one or more of the attributes requested. For example, a .bss section in `a.out` may not have the `SEC_HAS_CONTENTS` field set.

2.4.5.5 bfd_map_over_sections

Synopsis

```
void bfd_map_over_sections(bfd *abfd,
    void (*func)(bfd *abfd,
    asection *sect,
    PTR obj),
    PTR obj);
```

Description

Calls the provided function *func* for each section attached to the BFD *abfd*, passing *obj* as an argument. The function will be called as if by

```
func(abfd, the_section, obj);
```

This is the prefered method for iterating over sections, an alternative would be to use a loop:

```
section *p;
for (p = abfd->sections; p != NULL; p = p->next)
   func(abfd, p, ...)
```

2.4.5.6 bfd_set_section_size

Synopsis

```
boolean bfd_set_section_size(bfd *, asection *, bfd_size_type val);
```

Description

Sets *section* to the size *val*. If the operation is ok, then `true` is returned, else `false`. Possible error returns:

- invalid_operation Writing has started to the BFD, so setting the size is invalid

2.4.5.7 bfd_set_section_contents

Synopsis

```
boolean bfd_set_section_contents
   (bfd *abfd,
    asection *section,
    PTR data,
```

```
        file_ptr offset,
        bfd_size_type count);
```

Description

Sets the contents of the section *section* in BFD *abfd* to the data starting in memory at *data*. The data is written to the output section starting at offset *offset* for *count* bytes. Normally `true` is returned, else `false`. Possible error returns are:

- no_contents The output section does not have the `SEC_HAS_CONTENTS` attribute, so nothing can be written to it.
- and some more too This routine is front end to the back end function `_bfd_set_section_contents`.

2.4.5.8 `bfd_get_section_contents`

Synopsis

```
    boolean bfd_get_section_contents
       (bfd *abfd, asection *section, PTR location,
        file_ptr offset, bfd_size_type count);
```

Description

This function reads data from *section* in BFD *abfd* into memory starting at *location*. The data is read at an offset of *offset* from the start of the input section, and is read for *count* bytes. If the contents of a constuctor with the `SEC_CONSTUCTOR` flag set are requested, then the *location* is filled with zeroes. If no errors occur, `true` is returned, else `false`.

2.5 Symbols

BFD trys to maintain as much symbol information as it can when it moves information from file to file. BFD passes information to applications though the `asymbol` structure. When the application requests the symbol table, BFD reads the table in the native form and translates parts of it into the internal format. To maintain more than the infomation passed to applications some targets keep some information 'behind the sceans', in a structure only the particular back end knows about. For example, the coff back end keeps the original symbol table structure as well as the canonical structure when a BFD is read in. On output, the coff back end can reconstruct the output symbol table so that no information is lost, even information unique to coff which BFD doesn't know or

understand. If a coff symbol table was read, but was written through an a.out back end, all the coff specific information would be lost. The symbol table of a BFD is not necessarily read in until a canonicalize request is made. Then the BFD back end fills in a table provided by the application with pointers to the canonical information. To output symbols, the application provides BFD with a table of pointers to pointers to `asymbols`. This allows applications like the linker to output a symbol as read, since the 'behind the sceens' information will be still available.

2.5.1 Reading Symbols

There are two stages to reading a symbol table from a BFD; allocating storage, and the actual reading process. This is an excerpt from an appliction which reads the symbol table:

```
unsigned int storage_needed;
asymbol **symbol_table;
unsigned int number_of_symbols;
unsigned int i;

storage_needed = get_symtab_upper_bound (abfd);

if (storage_needed == 0) {
   return ;
}
symbol_table = (asymbol **) bfd_xmalloc (storage_needed);
  ...
number_of_symbols =
   bfd_canonicalize_symtab (abfd, symbol_table);

for (i = 0; i < number_of_symbols; i++) {
   process_symbol (symbol_table[i]);
}
```

All storage for the symbols themselves is in an obstack connected to the BFD, and is freed when the BFD is closed.

2.5.2 Writing Symbols

Writing of a symbol table is automatic when a BFD open for writing is closed. The application attaches a vector of pointers to pointers to symbols to the BFD being written, and fills in the symbol count. The close and cleanup code reads through the table provided and performs all the necessary operations. The outputing code must always be provided with an 'owned' symbol; one

which has come from another BFD, or one which has been created using `bfd_make_empty_symbol`. An example showing the creation of a symbol table with only one element:

```
#include "bfd.h"
main()
{
  bfd *abfd;
  asymbol *ptrs[2];
  asymbol *new;

  abfd = bfd_openw("foo","a.out-sunos-big");
  bfd_set_format(abfd, bfd_object);
  new = bfd_make_empty_symbol(abfd);
  new->name = "dummy_symbol";
  new->section = bfd_make_section_old_way(abfd, ".text");
  new->flags = BSF_GLOBAL;
  new->value = 0x12345;

  ptrs[0] = new;
  ptrs[1] = (asymbol *)0;

  bfd_set_symtab(abfd, ptrs, 1);
  bfd_close(abfd);
}

./makesym
nm foo
00012345 A dummy_symbol
```

Many formats cannot represent arbitary symbol information; for instance the `a.out` object format does not allow an arbitary number of sections. A symbol pointing to a section which is not one of `.text`, `.data` or `.bss` cannot be described.

2.5.3 typedef asymbol

An `asymbol` has the form:

.typedef struct symbol_cache_entry

```
{
/* A pointer to the BFD which owns the symbol. This information
   is necessary so that a back end can work out what additional
       (invisible to the application writer) information is carried
   with the symbol.  */

  struct _bfd *the_bfd;

/* The text of the symbol. The name is left alone, and not copied - the
   application may not alter it. */
```

```
  CONST char *name;

/* The value of the symbol.*/
  symvalue value;

/* Attributes of a symbol: */

#define BSF_NO_FLAGS    0x00

/* The symbol has local scope; <<static>> in <<C>>. The value
     is the offset into the section of the data. */
#define BSF_LOCAL 0x01

/* The symbol has global scope; initialized data in <<C>>. The
   value is the offset into the section of the data. */
#define BSF_GLOBAL 0x02

/* Obsolete */
#define BSF_IMPORT 0x04

/* The symbol has global scope, and is exported. The value is
   the offset into the section of the data. */
#define BSF_EXPORT 0x08

/* The symbol is undefined. <<extern>> in <<C>>. The value has
   no meaning. */
#define BSF_UNDEFINED_OBS 0x10

/* The symbol is common, initialized to zero; default in
   <<C>>. The value is the size of the object in bytes. */
#define BSF_FORT_COMM_OBS 0x20

/* A normal C symbol would be one of:
   <<BSF_LOCAL>>, <<BSF_FORT_COMM>>,  <<BSF_UNDEFINED>> or
   <<BSF_EXPORT|BSD_GLOBAL>> */

/* The symbol is a debugging record. The value has an arbitary
   meaning. */
#define BSF_DEBUGGING 0x40

/* Used by the linker */
#define BSF_KEEP        0x10000
#define BSF_KEEP_G      0x80000

/* Unused */
#define BSF_WEAK        0x100000
#define BSF_CTOR        0x200000

        /* This symbol was created to point to a section */
#define BSF_SECTION_SYM 0x400000
```

```
/* The symbol used to be a common symbol, but now it is
   allocated. */
#define BSF_OLD_COMMON  0x800000

/* The default value for common data. */
#define BFD_FORT_COMM_DEFAULT_VALUE 0

/* In some files the type of a symbol sometimes alters its
   location in an output file - ie in coff a <<ISFCN>> symbol
   which is also <<C_EXT>> symbol appears where it was
   declared and not at the end of a section.  This bit is set
      by the target BFD part to convey this information. */

#define BSF_NOT_AT_END    0x40000

/* Signal that the symbol is the label of constructor section. */
#define BSF_CONSTRUCTOR   0x1000000

/* Signal that the symbol is a warning symbol. If the symbol
   is a warning symbol, then the value field (I know this is
   tacky) will point to the asymbol which when referenced will
   cause the warning. */
#define BSF_WARNING       0x2000000

/* Signal that the symbol is indirect. The value of the symbol
   is a pointer to an undefined asymbol which contains the
   name to use instead. */
#define BSF_INDIRECT     0x4000000

  flagword flags;

/* A pointer to the section to which this symbol is
   relative.  This will always be non NULL, there are special
          sections for undefined and absolute symbols */
  struct sec *section;

/* Back end special data. This is being phased out in favour
   of making this a union. */
  PTR udata;

} asymbol;
```

2.5.4 Symbol Handling Functions

2.5.4.1 get_symtab_upper_bound

Description
Returns the number of bytes required in a vector of pointers to `asymbols` for all the symbols in the supplied BFD, including a terminal NULL pointer. If there are no symbols in the BFD, then 0 is returned.

```
#define get_symtab_upper_bound(abfd) \
     BFD_SEND (abfd, _get_symtab_upper_bound, (abfd))
```

2.5.4.2 bfd_canonicalize_symtab

Description
Supplied a BFD and a pointer to an uninitialized vector of pointers. This reads in the symbols from the BFD, and fills in the table with pointers to the symbols, and a trailing NULL. The routine returns the actual number of symbol pointers not including the NULL.

```
#define bfd_canonicalize_symtab(abfd, location) \
     BFD_SEND (abfd, _bfd_canonicalize_symtab,\
                  (abfd, location))
```

2.5.4.3 bfd_set_symtab

Description
Provided a table of pointers to symbols and a count, writes to the output BFD the symbols when closed.
Synopsis

```
boolean bfd_set_symtab (bfd *, asymbol **, unsigned int );
```

2.5.4.4 bfd_print_symbol_vandf

Description
Prints the value and flags of the symbol supplied to the stream file.
Synopsis

```
void bfd_print_symbol_vandf(PTR file, asymbol *symbol);
```

2.5.4.5 bfd_make_empty_symbol

Description
This function creates a new `asymbol` structure for the BFD, and returns a pointer to it. This routine is necessary, since each back end has private information surrounding the `asymbol`. Building your own `asymbol` and pointing to it will not create the private information, and will cause problems later on.

```
#define bfd_make_empty_symbol(abfd) \
     BFD_SEND (abfd, _bfd_make_empty_symbol, (abfd))
```

2.5.4.6 bfd_decode_symclass

Description
Return a lower-case character corresponding to the symbol class of symbol.
Synopsis

```
int bfd_decode_symclass(asymbol *symbol);
```

2.6 Archives

Description
Archives are supported in BFD in `archive.c`. An archive (or library) is just another BFD. It has a symbol table, although there's not much a user program will do with it. The big difference between an archive BFD and an ordinary BFD is that the archive doesn't have sections. Instead it has a chain of BFDs considered its contents. These BFDs can be manipulated just like any other. The BFDs contained in an archive opened for reading will all be opened for reading; you may put either input or output BFDs into an archive opened for output; it will be handled correctly when the archive is closed. Use `bfd_openr_next_archived_file` to step through all the contents of an archive opened for input. It's not required that you read the entire archive if you don't want to! Read it until you find what you want. Archive contents of output BFDs are chained through the `next` pointer in a BFD. The first one is findable through the `archive_head` slot of the archive. Set it with `set_archive_head` (q.v.). A given BFD may be in only one open output archive at a time. As expected, the BFD archive code is more general than the archive code of any given environment. BFD archives may contain files of different formats (eg a.out and coff) and even different architectures. You may even place archives recursively into archives! This can cause

unexpected confusion, since some archive formats are more expressive than others. For instance intel COFF archives can preserve long filenames; Sun a.out archives cannot. If you move a file from the first to the second format and back again, the filename may be truncated. Likewise, different a.out environments have different conventions as to how they truncate filenames, whether they preserve directory names in filenames, etc. When interoperating with native tools, be sure your files are homogeneous. Beware: most of these formats do not react well to the presence of spaces in filenames. We do the best we can, but can't always handle this due to restrctions in the format of archives. Many unix utilities are braindead in regards to spaces and such in filenames anyway, so this shouldn't be much of a restriction.

2.6.0.1 bfd_get_next_mapent

Synopsis

```
symindex bfd_get_next_mapent(bfd *, symindex previous, carsym ** sym);
```

Description

This function steps through an archive's symbol table (if it has one). Successively updates `sym` with the next symbol's information, returning that symbol's (internal) index into the symbol table. Supply BFD_NO_MORE_SYMBOLS as the `previous` entry to get the first one; returns BFD_NO_MORE_SYMBOLS when you're already got the last one. A `carsym` is a canonical archive symbol. The only user-visible element is its name, a null-terminated string.

2.6.0.2 bfd_set_archive_head

Synopsis

```
boolean bfd_set_archive_head(bfd *output, bfd *new_head);
```

Description

Used whilst processing archives. Sets the head of the chain of BFDs contained in an archive to *new_head*.

2.6.0.3 bfd_get_elt_at_index

Synopsis

```
bfd *bfd_get_elt_at_index(bfd * archive, int index);
```

Description
Return the bfd which is referenced by the symbol indexed by `index`. `index` should have been returned by `bfd_get_next_mapent` (q.v.).

2.6.0.4 `bfd_openr_next_archived_file`

Synopsis

```
bfd* bfd_openr_next_archived_file(bfd *archive, bfd *previous);
```

Description
Initially provided a BFD containing an archive and NULL, opens an inpout BFD on the first contained element and returns that. Subsequent calls to bfd_openr_next_archived_file should pass the archive and the previous return value to return a created BFD to the next contained element. NULL is returned when there are no more.

2.7 File Formats

A format is a BFD concept of high level file contents. The formats supported by BFD are:

- bfd_object
  The BFD may contain data, symbols, relocations and debug info.
- bfd_archive The BFD contains other BFDs and an optional index.
- bfd_core The BFD contains the result of an executable core dump.

2.7.0.1 `bfd_check_format`

Synopsis

```
boolean bfd_check_format(bfd *abfd, bfd_format format);
```

Description
This routine is supplied a BFD and a format. It attempts to verify if the file attached to the BFD is indeed compatible with the format specified (ie, one of `bfd_object`, `bfd_archive` or `bfd_core`).

If the BFD has been set to a specific *target* before the call, only the named target and format combination will be checked. If the target has not been set, or has been set to `default` then all the known target backends will be interrogated to determine a match. The function returns `true` on success, otherwise `false` with one of the following error codes:

- invalid_operation if `format` is not one of `bfd_object`, `bfd_archive`or `bfd_core`.
- system_call_error if an error occured during a read - even some file mismatches can cause system_call_errros
- file_not_recognised none of the backends recognised the file format
- file_ambiguously_recognized more than one backend recognised the file format.

2.7.0.2 `bfd_set_format`

Synopsis

```
boolean bfd_set_format(bfd *, bfd_format);
```

Description

This function sets the file format of the supplied BFD to the format requested. If the target set in the BFD does not support the format requested, the format is illegal or the BFD is not open for writing than an error occurs.

2.7.0.3 `bfd_format_string`

Synopsis

```
CONST char *bfd_format_string(bfd_format);
```

Description

This function takes one argument, and enumerated type (bfd_format) and returns a pointer to a const string `invalid`, `object`, `archive`, `core` or `unknown` depending upon the value of the enumeration.

2.8 Relocations

BFD maintains relocations in much the same was as it maintains symbols; they are left alone until required, then read in en-mass and traslated into an internal form. There is a common routine `bfd_perform_relocation` which acts upon the canonical form to to the actual fixup. Note that relocations are maintained on a per section basis, whilst symbols are maintained on a per BFD basis. All a back end has to do to fit the BFD interface is to create as many `struct reloc_cache_entry` as there are relocations in a particuar section, and fill in the right bits:

2.8.1 typedef arelent

This is the structure of a relocation entry:

.

```
typedef enum bfd_reloc_status
{
        /* No errors detected */
  bfd_reloc_ok,

        /* The relocation was performed, but there was an overflow. */
  bfd_reloc_overflow,

        /* The address to relocate was not within the section supplied*/
  bfd_reloc_outofrange,

        /* Used by special functions */
  bfd_reloc_continue,

        /* Unused */
  bfd_reloc_notsupported,

        /* Unsupported relocation size requested.  */
  bfd_reloc_other,

        /* The symbol to relocate against was undefined.*/
  bfd_reloc_undefined,

        /* The relocation was performed, but may not be ok - presently
          generated only when linking i960 coff files with i960 b.out
          symbols. */
  bfd_reloc_dangerous
 }
```

```
 bfd_reloc_status_type;


typedef struct reloc_cache_entry
{
        /* A pointer into the canonical table of pointers  */
  struct symbol_cache_entry **sym_ptr_ptr;

        /* offset in section */
  rawdata_offset address;

        /* addend for relocation value */
  bfd_vma addend;

        /* Pointer to how to perform the required relocation */
  CONST struct reloc_howto_struct *howto;

} arelent;
```

Description

Here is a description of each of the fields within a relent:

- sym_ptr_ptr

  The symbol table pointer points to a pointer to the symbol associated with the relocation request. This would naturally be the pointer into the table returned by the back end's get_symtab action. See ⟨undefined⟩ [Symbols], page ⟨undefined⟩. The symbol is referenced through a pointer to a pointer so that tools like the linker can fix up all the symbols of the same name by modifying only one pointer. The relocation routine looks in the symbol and uses the base of the section the symbol is attached to and the value of the symbol as the initial relocation offset. If the symbol pointer is zero, then the section provided is looked up.

- address The address field gives the offset in bytes from the base of the section data which owns the relocation record to the first byte of relocatable information. The actual data relocated will be relative to this point - for example, a relocation type which modifies the bottom two bytes of a four byte word would not touch the first byte pointed to in a big endian world.

- addend The addend is a value provided by the back end to be added (!) to the relocation offset. Its interpretation is dependent upon the howto. For example, on the 68k the code:

```
char foo[];
main()
        {
        return foo[0x12345678];
        }
```

  Could be compiled into:

```
linkw fp,#-4
moveb @#12345678,d0
extbl d0
unlk fp
```

```
        rts
```

This could create a reloc pointing to foo, but leave the offset in the data (something like)

```
RELOCATION RECORDS FOR [.text]:
offset   type      value
00000006 32        _foo

00000000 4e56 fffc          ; linkw fp,#-4
00000004 1039 1234 5678     ; moveb @#12345678,d0
0000000a 49c0               ; extbl d0
0000000c 4e5e               ; unlk fp
0000000e 4e75               ; rts
```

Using coff and an 88k, some instructions don't have enough space in them to represent the full address range, and pointers have to be loaded in two parts. So you'd get something like:

```
        or.u     r13,r0,hi16(_foo+0x12345678)
        ld.b     r2,r13,lo16(_foo+0x12345678)
        jmp      r1
```

This whould create two relocs, both pointing to _foo, and with 0x12340000 in their addend field. The data would consist of:

```
RELOCATION RECORDS FOR [.text]:
offset   type      value
00000002 HVRT16    _foo+0x12340000
00000006 LVRT16    _foo+0x12340000
00000000 5da05678           ; or.u r13,r0,0x5678
00000004 1c4d5678           ; ld.b r2,r13,0x5678
00000008 f400c001           ; jmp r1
```

The relocation routine digs out the value from the data, adds it to the addend to get the original offset and then adds the value of _foo. Note that all 32 bits have to be kept around somewhere, to cope with carry from bit 15 to bit 16. On further example is the sparc and the a.out format. The sparc has a similar problem to the 88k, in that some instructions don't have room for an entire offset, but on the sparc the parts are created odd sized lumps. The designers of the a.out format chose not to use the data within the section for storing part of the offset; all the offset is kept within the reloc. Any thing in the data should be ignored.

```
        save %sp,-112,%sp
        sethi %hi(_foo+0x12345678),%g2
        ldsb [%g2+%lo(_foo+0x12345678)],%i0
        ret
        restore
```

Both relocs contains a pointer to foo, and the offsets would contain junk.

```
RELOCATION RECORDS FOR [.text]:
offset   type      value
00000004 HI22      _foo+0x12345678
00000008 LO10      _foo+0x12345678
00000000 9de3bf90     ; save %sp,-112,%sp
00000004 05000000     ; sethi %hi(_foo+0),%g2
00000008 f048a000     ; ldsb [%g2+%lo(_foo+0)],%i0
0000000c 81c7e008     ; ret
```

```
00000010 81e80000      ; restore
```

- howto

  The howto field can be imagined as a relocation instruction. It is a pointer to a struct which contains information on what to do with all the other information in the reloc record and data section. A back end would normally have a relocation instruction set and turn relocations into pointers to the correct structure on input - but it would be possible to create each howto field on demand.

2.8.1.1 reloc_howto_type

The reloc_howto_type is a structure which contains all the information that BFD needs to know to tie up a back end's data.
.struct symbol_cache_entry; /* Forward declaration */

```
typedef CONST struct reloc_howto_struct
{
        /*  The type field has mainly a documetary use - the back end can
           to what it wants with it, though the normally the back end's
           external idea of what a reloc number would be would be stored
           in this field. For example, the a PC relative word relocation
           in a coff environment would have the type 023 - because that's
           what the outside world calls a R_PCRWORD reloc. */
  unsigned int type;

        /*  The value the final relocation is shifted right by. This drops
           unwanted data from the relocation.  */
  unsigned int rightshift;

        /*  The size of the item to be relocated - 0, is one byte, 1 is 2
           bytes, 3 is four bytes. */
  unsigned int size;

        /*  Now obsolete */
  unsigned int bitsize;

        /*  Notes that the relocation is relative to the location in the
           data section of the addend. The relocation function will
           subtract from the relocation value the address of the location
           being relocated. */
  boolean pc_relative;
```

```
      /*  Now obsolete */
unsigned int bitpos;

      /*  Now obsolete */
boolean absolute;

      /* Causes the relocation routine to return an error if overflow
        is detected when relocating. */
boolean complain_on_overflow;

      /* If this field is non null, then the supplied function is
        called rather than the normal function. This allows really
        strange relocation methods to be accomodated (eg, i960 callj
        instructions). */
bfd_reloc_status_type EXFUN ((*special_function),
  (bfd *abfd,
   arelent *reloc_entry,
                                        struct symbol_cache_entry *symbol,
                                        PTR data,
                                        asection *input_section));

      /* The textual name of the relocation type. */
char *name;

      /* When performing a partial link, some formats must modify the
        relocations rather than the data - this flag signals this.*/
boolean partial_inplace;

      /* The src_mask is used to select what parts of the read in data
        are to be used in the relocation sum. Eg, if this was an 8 bit
        bit of data which we read and relocated, this would be
        0x000000ff. When we have relocs which have an addend, such as
        sun4 extended relocs, the value in the offset part of a
        relocating field is garbage so we never use it. In this case
        the mask would be 0x00000000. */
bfd_word src_mask;

      /* The dst_mask is what parts of the instruction are replaced
        into the instruction. In most cases src_mask == dst_mask,
        except in the above special case, where dst_mask would be
        0x000000ff, and src_mask would be 0x00000000.   */
bfd_word dst_mask;

      /* When some formats create PC relative instructions, they leave
        the value of the pc of the place being relocated in the offset
        slot of the instruction, so that a PC relative relocation can
        be made just by adding in an ordinary offset (eg sun3 a.out).
        Some formats leave the displacement part of an instruction
        empty (eg m88k bcs), this flag signals the fact.*/
boolean pcrel_offset;
```

```
} reloc_howto_type;
```

2.8.1.2 the HOWTO macro

Description
The HOWTO define is horrible and will go away.

```
#define HOWTO(C, R,S,B, P, BI, ABS, O, SF, NAME, INPLACE, MASKSRC, MASKDST, PC)
\
  {(unsigned)C,R,S,B, P, BI, ABS,O,SF,NAME,INPLACE,MASKSRC,MASKDST,PC}
```

Description
And will be replaced with the totally magic way. But for the moment, we are compatible, so do it this way..

```
#define NEWHOWTO( FUNCTION, NAME,SIZE,REL,IN) HOWTO(0,0,SIZE,0,REL,0,false,false,FUNCTION
NAME,false,0,0,IN)
```

Description
Helper routine to turn a symbol into a relocation value.

```
#define HOWTO_PREPARE(relocation, symbol)      \
  {                                            \
  if (symbol != (asymbol *)NULL) {             \
    if (symbol->section == &bfd_com_section) { \
      relocation = 0;                          \
    }                                          \
    else {                                     \
      relocation = symbol->value;              \
    }                                          \
  }                                            \
}
```

2.8.1.3 reloc_chain

Description
How relocs are tied together

```
typedef unsigned char bfd_byte;

typedef struct relent_chain {
  arelent relent;
  struct   relent_chain *next;
```

```
} arelent_chain;
```

2.8.1.4 bfd_perform_relocation

Synopsis

```
bfd_reloc_status_type
bfd_perform_relocation
   (bfd * abfd,
    arelent *reloc_entry,
    PTR data,
    asection *input_section,
    bfd *output_bfd);
```

Description

If an output_bfd is supplied to this function the generated image will be relocatable, the relocations are copied to the output file after they have been changed to reflect the new state of the world. There are two ways of reflecting the results of partial linkage in an output file; by modifying the output data in place, and by modifying the relocation record. Some native formats (eg basic a.out and basic coff) have no way of specifying an addend in the relocation type, so the addend has to go in the output data. This is no big deal since in these formats the output data slot will always be big enough for the addend. Complex reloc types with addends were invented to solve just this problem.

2.9 The howto manager

When an application wants to create a relocation, but doesn't know what the target machine might call it, it can find out by using this bit of code.

2.9.0.1 bfd_reloc_code_type

Description

The insides of a reloc code

.

```
typedef enum bfd_reloc_code_real
```

```
{
       /* 16 bits wide, simple reloc */
  BFD_RELOC_16,

       /* 8 bits wide, but used to form an address like 0xffnn */
  BFD_RELOC_8_FFnn,

       /* 8 bits wide, simple */
  BFD_RELOC_8,

       /* 8 bits wide, pc relative */
  BFD_RELOC_8_PCREL,

       /* The type of reloc used to build a contructor table - at the
          moment probably a 32 bit wide abs address, but the cpu can
          choose. */

  BFD_RELOC_CTOR
 } bfd_reloc_code_real_type;
```

2.10 bfd_reloc_type_lookup

Synopsis

```
CONST struct reloc_howto_struct *
bfd_reloc_type_lookup
   (CONST bfd_arch_info_type *arch, bfd_reloc_code_type code);
```

Description

This routine returns a pointer to a howto struct which when invoked, will perform the supplied relocation on data from the architecture noted.

2.10.0.1 bfd_default_reloc_type_lookup

Synopsis

```
CONST struct reloc_howto_struct *bfd_default_reloc_type_lookup
   (CONST struct bfd_arch_info *,
    bfd_reloc_code_type  code);
```

Description

Provides a default relocation lookuperer for any architectue

2.10.0.2 bfd_generic_relax_section

Synopsis

```
boolean bfd_generic_relax_section
   (bfd *abfd,
    asection *section,
    asymbol **symbols);
```

Description

Provides default handling for relaxing for back ends which don't do relaxing - ie does nothing

2.10.0.3 bfd_generic_get_relocated_section_contents

Synopsis

```
bfd_byte *
bfd_generic_get_relocated_section_contents(bfd *abfd,
    struct bfd_seclet_struct  *seclet)
```

Description

Provides default handling of relocation effort for back ends which can't be bothered to do it efficiently.

2.11 Core files

Description

Buff output this facinating topic

2.11.0.1 bfd_core_file_failing_command

Synopsis

```
CONST char *bfd_core_file_failing_command(bfd *);
```

Description

Returns a read-only string explaining what program was running when it failed and produced the core file being read

2.11.0.2 bfd_core_file_failing_signal

Synopsis

```
int bfd_core_file_failing_signal(bfd *);
```

Description

Returns the signal number which caused the core dump which generated the file the BFD is attached to.

2.11.0.3 core_file_matches_executable_p

Synopsis

```
boolean core_file_matches_executable_p
   (bfd *core_bfd, bfd *exec_bfd);
```

Description

Returns `true` if the core file attached to *core_bfd* was generated by a run of the executable file attached to *exec_bfd*, or else `false`.

2.12 Targets

Description

Each port of BFD to a different machine requries the creation of a target back end. All the back end provides to the root part of BFD is a structure containing pointers to functions which perform certain low level operations on files. BFD translates the applications's requests through a pointer into calls to the back end routines. When a file is opened with `bfd_openr`, its format and target are unknown. BFD uses various mechanisms to determine how to interpret the file. The operations performed are:

- First a BFD is created by calling the internal routine `new_bfd`, then `bfd_find_target` is called with the target string supplied to `bfd_openr` and the new BFD pointer.
- If a null target string was provided to `bfd_find_target`, it looks up the environment variable `GNUTARGET` and uses that as the target string.
- If the target string is still NULL, or the target string is `default`, then the first item in the target vector is used as the target type. See ⟨undefined⟩ [bfd˙target], page ⟨undefined⟩.

- Otherwise, the elements in the target vector are inspected one by one, until a match on target name is found. When found, that is used.
- Otherwise the error `invalid_target` is returned to `bfd_openr`.
- `bfd_openr` attempts to open the file using `bfd_open_file`, and returns the BFD. Once the BFD has been opened and the target selected, the file format may be determined. This is done by calling `bfd_check_format` on the BFD with a suggested format. The routine returns `true` when the application guesses right.

2.12.1 bfd_target

Description

This structure contains everything that BFD knows about a target. It includes things like its byte order, name, what routines to call to do various operations, etc. Every BFD points to a target structure with its `xvec` member. Shortcut for declaring fields which are prototyped function pointers, while avoiding anguish on compilers that don't support protos.

```
#define SDEF(ret, name, arglist) \
                PROTO(ret,(*name),arglist)
#define SDEF_FMT(ret, name, arglist) \
                PROTO(ret,(*name[bfd_type_end]),arglist)
```

These macros are used to dispatch to functions through the bfd_target vector. They are used in a number of macros further down in '`bfd.h`', and are also used when calling various routines by hand inside the BFD implementation. The "arglist" argument must be parenthesized; it contains all the arguments to the called function.

```
#define BFD_SEND(bfd, message, arglist) \
               ((*((bfd)->xvec->message)) arglist)
```

For operations which index on the BFD format

```
#define BFD_SEND_FMT(bfd, message, arglist) \
            (((bfd)->xvec->message[(int)((bfd)->format)]) arglist)
```

This is the struct which defines the type of BFD this is. The `xvec` member of the struct `bfd` itself points here. Each module that implements access to a different target under BFD, defines one of these. FIXME, these names should be rationalised with the names of the entry points which call them. Too bad we can't have one macro to define them both!

```
typedef struct bfd_target
{
```

identifies the kind of target, eg SunOS4, Ultrix, etc

```
  char *name;
```

The "flavour" of a back end is a general indication about the contents of a file.

```
enum target_flavour {
  bfd_target_unknown_flavour,
  bfd_target_aout_flavour,
  bfd_target_coff_flavour,
  bfd_target_elf_flavour,
  bfd_target_ieee_flavour,
  bfd_target_oasys_flavour,
  bfd_target_srec_flavour} flavour;
```

The order of bytes within the data area of a file.

```
boolean byteorder_big_p;
```

The order of bytes within the header parts of a file.

```
boolean header_byteorder_big_p;
```

This is a mask of all the flags which an executable may have set - from the set `NO_FLAGS`, `HAS_RELOC`, ...`D_PAGED`.

```
flagword object_flags;
```

This is a mask of all the flags which a section may have set - from the set `SEC_NO_FLAGS`, `SEC_ALLOC`, ...`SET_NEVER_LOAD`.

```
flagword section_flags;
```

The pad character for filenames within an archive header.

```
char ar_pad_char;
```

The maximum number of characters in an archive header.

```
unsigned short ar_max_namelen;
```

The minimum alignment restriction for any section.

```
unsigned int align_power_min;
```

Entries for byte swapping for data. These are different to the other entry points, since they don't take BFD as first arg. Certain other handlers could do the same.

```
SDEF (bfd_vma,      bfd_getx64, (bfd_byte *));
SDEF (void,         bfd_putx64, (bfd_vma, bfd_byte *));
SDEF (bfd_vma, bfd_getx32, (bfd_byte *));
SDEF (void,         bfd_putx32, (bfd_vma, bfd_byte *));
SDEF (bfd_vma, bfd_getx16, (bfd_byte *));
SDEF (void,         bfd_putx16, (bfd_vma, bfd_byte *));
```

Byte swapping for the headers

```
SDEF (bfd_vma,   bfd_h_getx64, (bfd_byte *));
SDEF (void,          bfd_h_putx64, (bfd_vma, bfd_byte *));
SDEF (bfd_vma,  bfd_h_getx32, (bfd_byte *));
SDEF (void,          bfd_h_putx32, (bfd_vma, bfd_byte *));
SDEF (bfd_vma,  bfd_h_getx16, (bfd_byte *));
SDEF (void,          bfd_h_putx16, (bfd_vma, bfd_byte *));
```

Format dependent routines, these turn into vectors of entry points within the target vector structure; one for each format to check. Check the format of a file being read. Return bfd_target * or zero.

```
SDEF_FMT (struct bfd_target *, _bfd_check_format, (bfd *));
```

Set the format of a file being written.

```
SDEF_FMT (boolean,            _bfd_set_format, (bfd *));
```

Write cached information into a file being written, at bfd_close.

```
SDEF_FMT (boolean,            _bfd_write_contents, (bfd *));
```

The following functions are defined in `JUMP_TABLE`. The idea is that the back end writer of `foo` names all the routines `foo_`*entry_point*, `JUMP_TABLE` will built the entries in this structure in the right order. Core file entry points

```
SDEF (char *, _core_file_failing_command, (bfd *));
SDEF (int,    _core_file_failing_signal, (bfd *));
SDEF (boolean, _core_file_matches_executable_p, (bfd *, bfd *));
```

Archive entry points

```
SDEF (boolean, _bfd_slurp_armap, (bfd *));
SDEF (boolean, _bfd_slurp_extended_name_table, (bfd *));
SDEF (void,   _bfd_truncate_arname, (bfd *, CONST char *, char *));
SDEF (boolean, write_armap, (bfd *arch,
                             unsigned int elength,
                             struct orl *map,
                             unsigned int orl_count,
                             int stridx));
```

Standard stuff.

```
SDEF (boolean, _close_and_cleanup, (bfd *));
SDEF (boolean, _bfd_set_section_contents, (bfd *, sec_ptr, PTR,
                                           file_ptr, bfd_size_type));
SDEF (boolean, _bfd_get_section_contents, (bfd *, sec_ptr, PTR,
                                           file_ptr, bfd_size_type));
SDEF (boolean, _new_section_hook, (bfd *, sec_ptr));
```

Symbols and reloctions

```
SDEF (unsigned int, _get_symtab_upper_bound, (bfd *));
SDEF (unsigned int, _bfd_canonicalize_symtab,
         (bfd *, struct symbol_cache_entry **));
SDEF (unsigned int, _get_reloc_upper_bound, (bfd *, sec_ptr));
SDEF (unsigned int, _bfd_canonicalize_reloc, (bfd *, sec_ptr, arelent **,
                                              struct symbol_cache_entry**));
SDEF (struct symbol_cache_entry  *, _bfd_make_empty_symbol, (bfd *));
SDEF (void,     _bfd_print_symbol, (bfd *, PTR, struct symbol_cache_entry  *,
                                    bfd_print_symbol_type));
#define bfd_print_symbol(b,p,s,e) BFD_SEND(b, _bfd_print_symbol, (b,p,s,e))
SDEF (alent *,   _get_lineno, (bfd *, struct symbol_cache_entry  *));

SDEF (boolean,   _bfd_set_arch_mach, (bfd *, enum bfd_architecture,
                                      unsigned long));

SDEF (bfd *,  openr_next_archived_file, (bfd *arch, bfd *prev));
SDEF (boolean, _bfd_find_nearest_line,
      (bfd *abfd, struct sec  *section,
       struct symbol_cache_entry  **symbols,bfd_vma offset,
      CONST char **file, CONST char **func, unsigned int *line));
```

```
  SDEF (int,    _bfd_stat_arch_elt, (bfd *, struct stat *));

  SDEF (int,    _bfd_sizeof_headers, (bfd *, boolean));

  SDEF (void, _bfd_debug_info_start, (bfd *));
  SDEF (void, _bfd_debug_info_end, (bfd *));
  SDEF (void, _bfd_debug_info_accumulate, (bfd *, struct sec  *));
  SDEF (bfd_byte *, _bfd_get_relocated_section_contents, (bfd*,struct bfd_seclet_struct
*));
  SDEF (boolean,_bfd_relax_section,(bfd *, struct sec *, struct symbol_cache_entry
**));
```

Special entry points for gdb to swap in coff symbol table parts

```
  SDEF(void, _bfd_coff_swap_aux_in,(
       bfd            *abfd ,
       PTR             ext,
       int             type,
       int             class ,
       PTR             in));

  SDEF(void, _bfd_coff_swap_sym_in,(
       bfd            *abfd ,
       PTR             ext,
       PTR             in));

  SDEF(void, _bfd_coff_swap_lineno_in,  (
       bfd            *abfd,
       PTR            ext,
       PTR             in));
```

Special entry points for gas to swap coff parts

```
 SDEF(unsigned int, _bfd_coff_swap_aux_out,(
       bfd    *abfd,
       PTR in,
       int     type,
       int     class,
       PTR     ext));

 SDEF(unsigned int, _bfd_coff_swap_sym_out,(
      bfd      *abfd,
      PTR in,
      PTR ext));

 SDEF(unsigned int, _bfd_coff_swap_lineno_out,(
       bfd    *abfd,
       PTR in,
PTR ext));

 SDEF(unsigned int, _bfd_coff_swap_reloc_out,(
       bfd     *abfd,
```

```
      PTR src,
PTR dst));

 SDEF(unsigned int, _bfd_coff_swap_filehdr_out,(
       bfd   *abfd,
PTR  in,
PTR  out));

 SDEF(unsigned int, _bfd_coff_swap_aouthdr_out,(
       bfd  *abfd,
PTR  in,
PTR out));

 SDEF(unsigned int, _bfd_coff_swap_scnhdr_out,(
       bfd   *abfd,
       PTR in,
PTR out));

} bfd_target;
```

2.12.1.1 bfd_find_target

Description
Returns a pointer to the transfer vector for the object target named target_name. If target_name is NULL, chooses the one in the environment variable GNUTARGET; if that is null or not defined thenthe first entry in the target list is chosen. Passing in the string "default" or setting the environment variable to "default" will cause the first entry in the target list to be returned, and "target_defaulted" will be set in the BFD. This causes `bfd_check_format` to loop over all the targets to find the one that matches the file being read.
Synopsis

```
bfd_target *bfd_find_target(CONST char *, bfd *);
```

2.12.1.2 bfd_target_list

Description
This function returns a freshly malloced NULL-terminated vector of the names of all the valid BFD targets. Do not modify the names
Synopsis

```
CONST char **bfd_target_list(void);
```

2.13 Architectures

BFD's idea of an architecture is implimented in `archures.c`. BFD keeps one atom in a BFD describing the architecture of the data attached to the BFD; a pointer to a `bfd_arch_info_type`. Pointers to structures can be requested independently of a bfd so that an architecture's information can be interrogated without access to an open bfd. The arch information is provided by each architecture package. The set of default architectures is selected by the #define `SELECT_ARCHITECTURES`. This is normally set up in the `config\/h\-` file of your choice. If the name is not defined, then all the architectures supported are included. When BFD starts up, all the architectures are called with an initialize method. It is up to the architecture back end to insert as many items into the list of arches as it wants to, generally this would be one for each machine and one for the default case (an item with a machine field of 0).

2.13.1 bfd_architecture

Description
This enum gives the object file's CPU architecture, in a global sense. E.g. what processor family does it belong to? There is another field, which indicates what processor within the family is in use. The machine gives a number which distingushes different versions of the architecture, containing for example 2 and 3 for Intel i960 KA and i960 KB, and 68020 and 68030 for Motorola 68020 and 68030.

```
enum bfd_architecture
{
  bfd_arch_unknown,    /* File arch not known */
  bfd_arch_obscure,    /* Arch known, not one of these */
  bfd_arch_m68k,       /* Motorola 68xxx */
  bfd_arch_vax,        /* DEC Vax */
  bfd_arch_i960,       /* Intel 960 */
     /* The order of the following is important.
       lower number indicates a machine type that
       only accepts a subset of the instructions
       available to machines with higher numbers.
       The exception is the "ca", which is
       incompatible with all other machines except
       "core". */

#define bfd_mach_i960_core      1
#define bfd_mach_i960_ka_sa     2
#define bfd_mach_i960_kb_sb     3
#define bfd_mach_i960_mc        4
#define bfd_mach_i960_xa        5
```

```
#define bfd_mach_i960_ca        6

  bfd_arch_a29k,       /* AMD 29000 */
  bfd_arch_sparc,      /* SPARC */
  bfd_arch_mips,       /* MIPS Rxxxx */
  bfd_arch_i386,       /* Intel 386 */
  bfd_arch_ns32k,      /* National Semiconductor 32xxx */
  bfd_arch_tahoe,      /* CCI/Harris Tahoe */
  bfd_arch_i860,       /* Intel 860 */
  bfd_arch_romp,       /* IBM ROMP PC/RT */
  bfd_arch_alliant,    /* Alliant */
  bfd_arch_convex,     /* Convex */
  bfd_arch_m88k,       /* Motorola 88xxx */
  bfd_arch_pyramid,    /* Pyramid Technology */
  bfd_arch_h8300,      /* Hitachi H8/300 */
  bfd_arch_rs6000,     /* IBM RS/6000 */
  bfd_arch_last
  };
```

2.13.2 bfd_arch_info

Description

This structure contains information on architectures for use within BFD.

```
typedef int bfd_reloc_code_type;

typedef struct bfd_arch_info
{
  int bits_per_word;
  int bits_per_address;
  int bits_per_byte;
  enum bfd_architecture arch;
  long mach;
  char *arch_name;
  CONST  char *printable_name;
  /* true if this is the default machine for the architecture */
 unsigned int section_align_power;
  boolean the_default;
  CONST struct bfd_arch_info * EXFUN((*compatible),
(CONST struct bfd_arch_info *a,
 CONST struct bfd_arch_info *b));

  boolean EXFUN((*scan),(CONST struct bfd_arch_info *,CONST char *));
  unsigned int EXFUN((*disassemble),(bfd_vma addr, CONST char *data,
     PTR stream));
```

```
  CONST struct reloc_howto_struct *EXFUN((*reloc_type_lookup),
    (CONST struct bfd_arch_info *,
    bfd_reloc_code_type  code));

  struct bfd_arch_info *next;

} bfd_arch_info_type;
```

2.13.2.1 bfd_printable_name

Synopsis

```
CONST char *bfd_printable_name(bfd *abfd);
```

Description

Return a printable string representing the architecture and machine from the pointer to the arch info structure

2.13.2.2 bfd_scan_arch

Synopsis

```
bfd_arch_info_type *bfd_scan_arch(CONST char *);
```

Description

This routine is provided with a string and tries to work out if bfd supports any cpu which could be described with the name provided. The routine returns a pointer to an arch_info structure if a machine is found, otherwise NULL.

2.13.2.3 bfd_arch_get_compatible

Synopsis

```
CONST bfd_arch_info_type *bfd_arch_get_compatible(
    CONST bfd *abfd,
    CONST bfd *bbfd);
```

Description

This routine is used to determine whether two BFDs' architectures and achine types are compatible. It calculates the lowest common denominator between the two architectures and machine types

implied by the BFDs and returns a pointer to an arch_info structure describing the compatible machine.

2.13.2.4 bfd_default_arch_struct

Description
The `bfd_default_arch_struct` is an item of `bfd_arch_info_type` which has been initialized to a fairly generic state. A BFD starts life by pointing to this structure, until the correct back end has determined the real architecture of the file.

```
extern bfd_arch_info_type bfd_default_arch_struct;
```

2.13.2.5 bfd_set_arch_info

Synopsis

```
void bfd_set_arch_info(bfd *, bfd_arch_info_type *);
```

2.13.2.6 bfd_default_set_arch_mach

Synopsis

```
boolean bfd_default_set_arch_mach(bfd *abfd,
    enum bfd_architecture arch,
    unsigned long mach);
```

Description
Set the architecture and machine type in a bfd. This finds the correct pointer to structure and inserts it into the arch_info pointer.

2.13.2.7 bfd_get_arch

Synopsis

```
enum bfd_architecture bfd_get_arch(bfd *abfd);
```

Description
Returns the enumerated type which describes the supplied bfd's architecture

2.13.2.8 bfd_get_mach

Synopsis

```
unsigned long bfd_get_mach(bfd *abfd);
```

Description

Returns the long type which describes the supplied bfd's machine

2.13.2.9 bfd_arch_bits_per_byte

Synopsis

```
unsigned int bfd_arch_bits_per_byte(bfd *abfd);
```

Description

Returns the number of bits in one of the architectures bytes

2.13.2.10 bfd_arch_bits_per_address

Synopsis

```
unsigned int bfd_arch_bits_per_address(bfd *abfd);
```

Description

Returns the number of bits in one of the architectures addresses

2.13.2.11 bfd_arch_init

Synopsis

```
void  bfd_arch_init(void);
```

Description

This routine initializes the architecture dispatch table by calling all installed architecture packages and getting them to poke around.

2.13.2.12 bfd_arch_linkin

Synopsis

```
void bfd_arch_linkin(bfd_arch_info_type *);
```

Description

Link the provided arch info structure into the list

2.13.2.13 bfd_default_compatible

Synopsis

```
CONST bfd_arch_info_type *bfd_default_compatible
   (CONST bfd_arch_info_type *a,
    CONST bfd_arch_info_type *b);
```

Description

The default function for testing for compatibility.

2.13.2.14 bfd_default_scan

Synopsis

```
boolean bfd_default_scan(CONST struct bfd_arch_info *, CONST char *);
```

Description

The default function for working out whether this is an architecture hit and a machine hit.

2.13.2.15 bfd_get_arch_info

Synopsis

```
bfd_arch_info_type * bfd_get_arch_info(bfd *);
```

2.13.2.16 bfd_lookup_arch

Synopsis

```
bfd_arch_info_type *bfd_lookup_arch
   (enum bfd_architecture
```

```
        arch,
        long machine);
```

Description
Look for the architecure info struct which matches the arguments given. A machine of 0 will match the machine/architecture structure which marks itself as the default.

2.13.2.17 bfd_printable_arch_mach

Synopsis

```
    CONST char * bfd_printable_arch_mach
       (enum bfd_architecture arch, unsigned long machine);
```

Description
Return a printable string representing the architecture and machine type. NB. The use of this routine is depreciated.

2.14 Opening and Closing BFDs

2.14.0.1 bfd_openr

Synopsis

```
    bfd *bfd_openr(CONST char *filename, CONST char*target);
```

Description
This function opens the file supplied (using `fopen`) with the target supplied, it returns a pointer to the created BFD. If NULL is returned then an error has occured. Possible errors are `no_memory`, `invalid_target` or `system_call` error.

2.14.0.2 bfd_fdopenr

Synopsis

```
bfd *bfd_fdopenr(CONST char *filename, CONST char *target, int fd);
```

Description

bfd_fdopenr is to bfd_fopenr much like fdopen is to fopen. It opens a BFD on a file already described by the *fd* supplied. Possible errors are no_memory, invalid_target and system_call error.

2.14.0.3 bfd_openw

Synopsis

```
bfd *bfd_openw(CONST char *filename, CONST char *target);
```

Description

Creates a BFD, associated with file *filename*, using the file format *target*, and returns a pointer to it. Possible errors are system_call_error, no_memory, invalid_target.

2.14.0.4 bfd_close

Synopsis

```
boolean bfd_close(bfd *);
```

Description

This function closes a BFD. If the BFD was open for writing, then pending operations are completed and the file written out and closed. If the created file is executable, then `chmod` is called to mark it as such. All memory attached to the BFD's obstacks is released.

Returns

`true` is returned if all is ok, otherwise `false`.

2.14.0.5 bfd_close_all_done

Synopsis

```
boolean bfd_close_all_done(bfd *);
```

Description

This function closes a BFD. It differs from `bfd_close` since it does not complete any pending operations. This routine would be used if the application had just used BFD for swapping and didn't want to use any of the writing code. If the created file is executable, then `chmod` is called to mark it as such. All memory attached to the BFD's obstacks is released.

Returns
`true` is returned if all is ok, otherwise `false`.
Synopsis

```
bfd_size_type bfd_alloc_size(bfd *abfd);
```

Description
Return the number of bytes in the obstacks connected to the supplied BFD.

2.14.0.6 bfd_create

Synopsis

```
bfd *bfd_create(CONST char *filename, bfd *template);
```

Description
This routine creates a new BFD in the manner of `bfd_openw`, but without opening a file. The new BFD takes the target from the target used by *template*. The format is always set to `bfd_object`.

2.14.0.7 bfd_alloc_by_size_t

Synopsis

```
PTR bfd_alloc_by_size_t(bfd *abfd, size_t wanted);
```

Description
This function allocates a block of memory in the obstack attatched to `abfd` and returns a pointer to it.

2.15 Constructors

Classes in C++ have 'constructors' and 'destructors'. These are functions which are called automatically by the language whenever data of a class is created or destroyed. Class data which is static data may also be have a type which requires 'construction', the contructor must be called before the data can be referenced, so the contructor must be called before the program begins. The common solution to this problem is for the compiler to call a magic function as the first statement `main`. This magic function, (often called `__main`) runs around calling the constructors for all the things needing it. With COFF the compile has a bargain with the linker et al. All constructors are given strange names, for example `__GLOBAL__$I$foo` might be the label of a contructor for the

class *foo*. The solution on unfortunate systems (most system V machines) is to perform a partial link on all the .o files, do an `nm` on the result, run `awk` or some such over the result looking for strange `__GLOBAL__$` symbols, generate a C program from this, compile it and link with the partially linked input. This process is usually called `collect`. Some versions of `a.out` use something called the `set_vector` mechanism. The constructor symbols are output from the compiler with a special stab code saying that they are constructors, and the linker can deal with them directly. BFD allows applications (ie the linker) to deal with constructor information independently of their external implimentation by providing a set of entry points for the indiviual object back ends to call which maintains a database of the contructor information. The application can interrogate the database to find out what it wants. The construction data essential for the linker to be able to perform its job are:

- asymbol The asymbol of the contructor entry point contains all the information necessary to call the function.
- table id The type of symbol, ie is it a contructor, a destructor or something else someone dreamed up to make our lives difficult. This module takes this information and then builds extra sections attached to the bfds which own the entry points. It creates these sections as if they were tables of pointers to the entry points, and builds relocation entries to go with them so that the tables can be relocated along with the data they reference. These sections are marked with a special bit (`SEC_CONSTRUCTOR`) which the linker notices and do with what it wants.

2.15.0.1 `bfd_constructor_entry`

Synopsis

```
void bfd_constructor_entry(bfd *abfd,
    asymbol **symbol_ptr_ptr,
    CONST char*type);
```

Description

This function is called with an a symbol describing the function to be called, an string which descibes the xtor type, eg something like "CTOR" or "DTOR" would be fine. And the bfd which owns the function. Its duty is to create a section called "CTOR" or "DTOR" or whatever if the bfd doesn't already have one, and grow a relocation table for the entry points as they accumulate.

2.16 libbfd

Description
This file contains various routines which are used within BFD. They are not intended for export, but are documented here for completeness.

2.16.0.1 bfd_xmalloc

Synopsis

```
PTR  bfd_xmalloc( bfd_size_type size);
```

Description
Like malloc, but exit if no more memory.

2.16.0.2 bfd_write_bigendian_4byte_int

Synopsis

```
void bfd_write_bigendian_4byte_int(bfd *abfd,  int i);
```

Description
Writes a 4 byte integer to the outputing bfd, in big endian mode regardless of what else is going on. This is usefull in archives.

2.16.0.3 bfd_put_size

2.16.0.4 bfd_get_size

Description
These macros as used for reading and writing raw data in sections; each access (except for bytes) is vectored through the target format of the BFD and mangled accordingly. The mangling performs any necessary endian translations and removes alignment restrictions.

```
#define bfd_put_8(abfd, val, ptr) \
                (*((char *)ptr) = (char)val)
```

```
#define bfd_get_8(abfd, ptr) \
                (*((char *)ptr))
#define bfd_put_16(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx16, (val,ptr))
#define bfd_get_16(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx16, (ptr))
#define bfd_put_32(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx32, (val,ptr))
#define bfd_get_32(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx32, (ptr))
#define bfd_put_64(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx64, (val, ptr))
#define bfd_get_64(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx64, (ptr))
```

2.16.0.5 bfd_h_put_size

2.16.0.6 bfd_h_get_size

Description

These macros have the same function as their `bfd_get_x` bretherin, except that they are used for removing information for the header records of object files. Believe it or not, some object files keep their header records in big endian order, and their data in little endan order.

```
#define bfd_h_put_8(abfd, val, ptr) \
                (*((char *)ptr) = (char)val)
#define bfd_h_get_8(abfd, ptr) \
                (*((char *)ptr))
#define bfd_h_put_16(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_h_putx16,(val,ptr))
#define bfd_h_get_16(abfd, ptr) \
                BFD_SEND(abfd, bfd_h_getx16,(ptr))
#define bfd_h_put_32(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_h_putx32,(val,ptr))
#define bfd_h_get_32(abfd, ptr) \
                BFD_SEND(abfd, bfd_h_getx32,(ptr))
#define bfd_h_put_64(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_h_putx64,(val, ptr))
#define bfd_h_get_64(abfd, ptr) \
                BFD_SEND(abfd, bfd_h_getx64,(ptr))
```

2.16.0.7 bfd_log2

Description
Return the log base 2 of the value supplied, rounded up. eg an arg of 1025 would return 11.
Synopsis

```
bfd_vma bfd_log2(bfd_vma x);
```

2.17 File Caching

The file caching mechanism is embedded within BFD and allows the application to open as many BFDs as it wants without regard to the underlying operating system's file descriptor limit (often as low as 20 open files). The module in `cache.c` maintains a least recently used list of `BFD_CACHE_MAX_OPEN` files, and exports the name `bfd_cache_lookup` which runs around and makes sure that the required BFD is open. If not, then it chooses a file to close, closes it and opens the one wanted, returning its file handle.

2.17.0.1 BFD_CACHE_MAX_OPEN macro

Description
The maxiumum number of files which the cache will keep open at one time.

```
#define BFD_CACHE_MAX_OPEN 10
```

2.17.0.2 bfd_last_cache

Synopsis

```
extern bfd *bfd_last_cache;
```

Description
Zero, or a pointer to the topmost BFD on the chain. This is used by the `bfd_cache_lookup` macro in '`libbfd.h`' to determine when it can avoid a function call.

2.17.0.3 bfd_cache_lookup

Description
Checks to see if the required BFD is the same as the last one looked up. If so then it can use the iostream in the BFD with impunity, since it can't have changed since the last lookup, otherwise it has to perform the complicated lookup function

```
#define bfd_cache_lookup(x) \
    ((x)==bfd_last_cache? \
      (FILE*)(bfd_last_cache->iostream): \
       bfd_cache_lookup_worker(x))
```

2.17.0.4 bfd_cache_init

Synopsis

```
void  bfd_cache_init (bfd *);
```

Description
Initialize a BFD by putting it on the cache LRU.

2.17.0.5 bfd_cache_close

Description
Remove the BFD from the cache. If the attached file is open, then close it too.
Synopsis

```
void bfd_cache_close (bfd *);
```

2.17.0.6 bfd_open_file

Description
Call the OS to open a file for this BFD. Returns the FILE * (possibly null) that results from this operation. Sets up the BFD so that future accesses know the file is open. If the FILE returned is null, then there is won't have been put in the cache, so it won't have to be removed from it.
Synopsis

```
FILE* bfd_open_file(bfd *);
```

2.17.0.7 `bfd_cache_lookup_worker`

Description
Called when the macro `bfd_cache_lookup` fails to find a quick answer. Finds a file descriptor for this BFD. If necessary, it open it. If there are already more than BFD_CACHE_MAX_OPEN files open, it trys to close one first, to avoid running out of file descriptors.
Synopsis

```
FILE *bfd_cache_lookup_worker(bfd *);
```

3 BFD back end

All of BFD lives in one directory.

3.1 a.out backends

Description
BFD supports a number of different flavours of a.out format, though the major differences are only the sizes of the structures on disk, and the shape of the relocation information. The support is split into a basic support file `aoutx.h` and other files which derive functions from the base. One derivation file is `aoutf1.h` (for a.out flavour 1), and adds to the basic a.out functions support for sun3, sun4, 386 and 29k a.out files, to create a target jump vector for a specific target. This information is further split out into more specific files for each machine, including `sunos.c` for sun3 and sun4, `newsos3.c` for the Sony NEWS, and `demo64.c` for a demonstration of a 64 bit a.out format. The base file `aoutx.h` defines general mechanisms for reading and writing records to and from disk, and various other methods which BFD requires. It is included by `aout32.c` and `aout64.c` to form the names aout_32_swap_exec_header_in, aout_64_swap_exec_header_in, etc. As an example, this is what goes on to make the back end for a sun4, from aout32.c

```
#define ARCH_SIZE 32
#include "aoutx.h"
```

Which exports names:

```
...
aout_32_canonicalize_reloc
aout_32_find_nearest_line
aout_32_get_lineno
aout_32_get_reloc_upper_bound
...
```

from sunos.c

```
#define ARCH 32
#define TARGET_NAME "a.out-sunos-big"
#define VECNAME    sunos_big_vec
#include "aoutf1.h"
```

requires all the names from aout32.c, and produces the jump vector

```
sunos_big_vec
```

The file host-aout.c is a special case. It is for a large set of hosts that use "more or less standard" a.out files, and for which cross-debugging is not interesting. It uses the standard 32-bit a.out support routines, but determines the file offsets and addresses of the text, data, and BSS sections, the machine architecture and machine type, and the entry point address, in a host-dependent

manner. Once these values have been determined, generic code is used to handle the object file. When porting it to run on a new system, you must supply:

```
HOST_PAGE_SIZE
HOST_SEGMENT_SIZE
HOST_MACHINE_ARCH       (optional)
HOST_MACHINE_MACHINE    (optional)
HOST_TEXT_START_ADDR
HOST_STACK_END_ADDR
```

in the file `../include/sys/h-XXX.h` (for your host). These values, plus the structures and macros defined in `a.out.h` on your host system, will produce a BFD target that will access ordinary a.out files on your host. To configure a new machine to use `host-aout.c`, specify:

```
TDEFAULTS = -DDEFAULT_VECTOR=host_aout_big_vec
TDEPFILES= host-aout.o trad-core.o
```

in the `config/mt-XXX` file, and modify configure.in to use the `mt-XXX` file (by setting `"bfd_target=XXX"`) when your configuration is selected.

3.1.1 relocations

Description
The file `aoutx.h` caters for both the *standard* and *extended* forms of a.out relocation records. The standard records are characterised by containing only an address, a symbol index and a type field. The extended records (used on 29ks and sparcs) also have a full integer for an addend.

3.1.2 Internal Entry Points

Description
`aoutx.h` exports several routines for accessing the contents of an a.out file, which are gathered and exported in turn by various format specific files (eg sunos.c).

3.1.2.1 aout_<size>_swap_exec_header_in

Description
Swaps the information in an executable header taken from a raw byte stream memory image, into

the internal exec_header structure.
void aout_<size>_swap_exec_header_in, (bfd *abfd, struct external_exec *raw_bytes, struct internal_exec *execp);

3.1.2.2 aout_<size>_swap_exec_header_out

Description
Swaps the information in an internal exec header structure into the supplied buffer ready for writing to disk.
void aout_<size>_swap_exec_header_out (bfd *abfd, struct internal_exec *execp, struct external_exec *raw_bytes);

3.1.2.3 aout_<size>_some_aout_object_p

Description
Some A.OUT variant thinks that the file whose format we're checking is an a.out file. Do some more checking, and set up for access if it really is. Call back to the calling environments "finish up" function just before returning, to handle any last-minute setup.
bfd_target *aout_<size>_some_aout_object_p (bfd *abfd, bfd_target *(*callback_to_real_object_p)());

3.1.2.4 aout_<size>_mkobject

Description
This routine initializes a BFD for use with a.out files.
boolean aout_<size>_mkobject, (bfd *);

3.1.2.5 aout_<size>_machine_type

Description
Keep track of machine architecture and machine type for a.out's. Return the machine_type for a particular arch&machine, or M_UNKNOWN if that exact arch&machine can't be represented in a.out format. If the architecture is understood, machine type 0 (default) should always be understood.

enum machine_type aout_<size>_machine_type (enum bfd_architecture arch, unsigned long machine));

3.1.2.6 aout_<size>_set_arch_mach

Description
Sets the architecture and the machine of the BFD to those values supplied. Verifies that the format can support the architecture required.
boolean aout_<size>_set_arch_mach, (bfd *, enum bfd_architecture, unsigned long machine));

3.1.2.7 aout_<size>new_section_hook

Description
Called by the BFD in response to a `bfd_make_section` request.
boolean aout_<size>_new_section_hook, (bfd *abfd, asection *newsect));

3.2 coff backends

BFD supports a number of different flavours of coff format. The major difference between formats are the sizes and alignments of fields in structures on disk, and the occasional extra field. Coff in all its varieties is implimented with a few common files and a number of implementation specific files. For example, The 88k bcs coff format is implemented in the file `coff-m88k.c`. This file `#includes` `coff-m88k.h` which defines the external structure of the coff format for the 88k, and `internalcoff.h` which defines the internal structure. `coff-m88k.c` also defines pthe relocations used by the 88k format See ⟨undefined⟩ [Relocations], page ⟨undefined⟩. Then the major portion of coff code is included (`coffcode.h`) which defines the methods used to act upon the types defined in `coff-m88k.h` and `internalcoff.h`. The Intel i960 processor version of coff is implemented in `coff-i960.c`. This file has the same structure as `coff-m88k.c`, except that it includes `coff-i960.h` rather than `coff-m88k.h`.

3.2.1 Porting To A New Version of Coff

The recommended method is to select from the existing implimentations the version of coff which is most like the one you want to use, for our purposes, we'll say that i386 coff is the one you select, and

that your coff flavour is called foo. Copy the `i386coff.c` to `foocoff.c`, copy `../include/i386coff.h` to `../include/foocoff.h` and add the lines to `targets.c` and `Makefile.in` so that your new back end is used. Alter the shapes of the structures in `../include/foocoff.h` so that they match what you need. You will probably also have to add `#ifdef`s to the code in `internalcoff.h` and `coffcode.h` if your version of coff is too wild. You can verify that your new BFD backend works quite simply by building `objdump` from the `binutils` directory, and making sure that its version of what's going on at your host systems idea (assuming it has the pretty standard coff dump utility (usually called `att-dump` or just `dump`)) are the same. Then clean up your code, and send what you've done to Cygnus. Then your stuff will be in the next release, and you won't have to keep integrating it.

3.2.2 How The Coff Backend Works

3.2.2.1 Bit Twiddling

Each flavour of coff supported in BFD has its own header file descibing the external layout of the structures. There is also an internal description of the coff layout (in `internalcoff.h`) file (). A major function of the coff backend is swapping the bytes and twiddling the bits to translate the external form of the structures into the normal internal form. This is all performed in the `bfd_swap_`*thing*`_`*direction* routines. Some elements are different sizes between different versions of coff, it is the duty of the coff version specific include file to override the definitions of various packing routines in `coffcode.h`. Eg the size of line number entry in coff is sometimes 16 bits, and sometimes 32 bits. `#defineing` `PUT_LNSZ_LNNO` and `GET_LNSZ_LNNO` will select the correct one. No doubt, some day someone will find a version of coff which has a varying field size not catered for at the moment. To port BFD, that person will have to add more `#defines`. Three of the bit twiddling routines are exported to `gdb`; `coff_swap_aux_in`, `coff_swap_sym_in` and `coff_swap_linno_in`. `GDB` reads the symbol table on its own, but uses BFD to fix things up. More of the bit twiddlers are exported for `gas`; `coff_swap_aux_out`, `coff_swap_sym_out`, `coff_swap_lineno_out`, `coff_swap_reloc_out`, `coff_swap_filehdr_out`, `coff_swap_aouthdr_out`, `coff_swap_scnhdr_out`. `Gas` currently keeps track of all the symbol table and reloc drudgery itself, thereby saving the internal BFD overhead, but uses BFD to swap things on the way out, making cross ports much safer. This also allows BFD (and thus the linker) to use the same header files as `gas`, which makes one avenue to disaster disappear.

3.2.2.2 Symbol Reading

The simple canonical form for symbols used by BFD is not rich enough to keep all the information available in a coff symbol table. The back end gets around this by keeping the original symbol table around, "behind the scenes". When a symbol table is requested (through a call to `bfd_canonicalize_symtab`, a request gets through to `get_normalized_symtab`. This reads the symbol table from the coff file and swaps all the structures inside into the internal form. It also fixes up all the pointers in the table (represented in the file by offsets from the first symbol in the table) into physical pointers to elements in the new internal table. This involves some work since the meanings of fields changes depending upon context; a field that is a pointer to another structure in the symbol table at one moment may be the size in bytes of a structure in the next. Another pass is made over the table. All symbols which mark file names (`C_FILE` symbols) are modified so that the internal string points to the value in the auxent (the real filename) rather than the normal text associated with the symbol (`".file"`). At this time the symbol names are moved around. Coff stores all symbols less than nine characters long physically within the symbol table, longer strings are kept at the end of the file in the string table. This pass moves all strings into memory, and replaces them with pointers to the strings. The symbol table is massaged once again, this time to create the canonical table used by the BFD application. Each symbol is inspected in turn, and a decision made (using the `sclass` field) about the various flags to set in the `asymbol` See ⟨undefined⟩ [Symbols], page ⟨undefined⟩. The generated canonical table shares strings with the hidden internal symbol table. Any linenumbers are read from the coff file too, and attached to the symbols which own the functions the linenumbers belong to.

3.2.2.3 Symbol Writing

Writing a symbol to a coff file which didn't come from a coff file will lose any debugging information. The `asymbol` structure remembers the BFD from which was born, and on output the back end makes sure that the same destination target as source target is present. When the symbols have come from a coff file then all the debugging information is preserved. Symbol tables are provided for writing to the back end in a vector of pointers to pointers. This allows applications like the linker to accumulate and output large symbol tables without having to do too much byte copying. This function runs through the provided symbol table and patches each symbol marked as a file place holder (`C_FILE`) to point to the next file place holder in the list. It also marks each `offset` field in the list with the offset from the first symbol of the current symbol. Another function of this procedure is to turn the canonical value form of BFD into the form used by coff. Internally, BFD expects symbol values to be offsets from a section base; so a symbol physically at 0x120, but in a section starting at 0x100, would have the value 0x20. Coff expects symbols to contain their final value, so symbols have their values changed at this point to reflect their sum with their owning

section. Note that this transformation uses the `output_section` field of the `asymbol`'s `asection` See ⟨undefined⟩ [Sections], page ⟨undefined⟩.

- coff_mangle_symbols This routine runs though the provided symbol table and uses the offsets generated by the previous pass and the pointers generated when the symbol table was read in to create the structured hierachy required by coff. It changes each pointer to a symbol to an index into the symbol table of the symbol being referenced.
- coff_write_symbols This routine runs through the symbol table and patches up the symbols from their internal form into the coff way, calls the bit twiddlers and writes out the tabel to the file.

3.2.2.4 `coff_symbol_type`

Description
The hidden information for an asymbol is described in a coff_ptr_struct, which is typedefed to a combined_entry_type

.

```
typedef struct coff_ptr_struct
{

        /* Remembers the offset from the first symbol in the file for
          this symbol. Generated by coff_renumber_symbols. */
unsigned int offset;

        /* Should the tag field of this symbol be renumbered.
          Created by coff_pointerize_aux. */
char fix_tag;

        /* Should the endidx field of this symbol be renumbered.
          Created by coff_pointerize_aux. */
char fix_end;

        /* The container for the symbol structure as read and translated
           from the file. */

union {
   union internal_auxent auxent;
   struct internal_syment syment;
 } u;
} combined_entry_type;
```

```
/* Each canonical asymbol really looks like this: */

typedef struct coff_symbol_struct
{
    /* The actual symbol which the rest of BFD works with */
asymbol symbol;

    /* A pointer to the hidden information for this symbol */
combined_entry_type *native;

    /* A pointer to the linenumber information for this symbol */
struct lineno_cache_entry *lineno;

    /* Have the line numbers been relocated yet ? */
boolean done_lineno;
} coff_symbol_type;
```

3.2.2.5 Writing Relocations

To write relocations, all the back end does is step though the canonical relocation table, and create an `internal_reloc`. The symbol index to use is removed from the `offset` field in the symbol table supplied, the address comes directly from the sum of the section base address and the relocation offset and the type is dug directly from the howto field. Then the `internal_reloc` is swapped into the shape of an `external_reloc` and written out to disk.

3.2.2.6 Reading Linenumbers

Creating the linenumber table is done by reading in the entire coff linenumber table, and creating another table for internal use. A coff line number table is structured so that each function is marked as having a line number of 0. Each line within the function is an offset from the first line in the function. The base of the line number information for the table is stored in the symbol associated with the function. The information is copied from the external to the internal table, and each symbol which marks a function is marked by pointing its... How does this work ?

3.2.2.7 Reading Relocations

Coff relocations are easily transformed into the internal BFD form (`arelent`). Reading a coff relocation table is done in the following stages:

- The entire coff relocation table is read into memory.
- Each relocation is processed in turn, first it is swapped from the external to the internal form.
- The symbol referenced in the relocation's symbol index is turned intoa pointer into the canonical symbol table. Note that this table is the same as the one returned by a call to `bfd_canonicalize_symtab`. The back end will call the routine and save the result if a canonicalization hasn't been done.
- The reloc index is turned into a pointer to a howto structure, in a back end specific way. For instance, the 386 and 960 use the `r_type` to directly produce an index into a howto table vector; the 88k subtracts a number from the `r_type` field and creates an addend field.

Index

(Index is nonexistent)

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt and
cmsl10 at 10.95pt
are used for emphasis.

Table of Contents

1 Introduction 1

1.1 History 1
1.2 How It Works 1
1.3 What BFD Version 1 Can Do 2
1.3.1 Information Loss 3
1.3.2 Mechanism 3

2 BFD front end 6

2.1 typedef bfd 6
2.1.0.1 bfd_get_reloc_upper_bound 9
2.1.0.2 bfd_canonicalize_reloc 9
2.1.0.3 bfd_set_file_flags 9
2.1.0.4 bfd_set_reloc 10
2.1.0.5 bfd_set_start_address 10
2.1.0.6 The bfd_get_mtime function 10
2.1.0.7 stuff 10
2.2 Memory Usage 12
2.3 Initialization 12
2.3.0.1 bfd_init 12
2.3.0.2 bfd_check_init 13
2.4 Sections 13
2.4.1 Section Input 13
2.4.2 Section Output 14
2.4.3 Seglets 14
2.4.4 typedef asection 14
2.4.5 section prototypes 19
2.4.5.1 bfd_get_section_by_name 19
2.4.5.2 bfd_make_section_old_way 19
2.4.5.3 bfd_make_section 20
2.4.5.4 bfd_set_section_flags 20
2.4.5.5 bfd_map_over_sections 21
2.4.5.6 bfd_set_section_size 21
2.4.5.7 bfd_set_section_contents 21
2.4.5.8 bfd_get_section_contents 22
2.5 Symbols 22
2.5.1 Reading Symbols 23
2.5.2 Writing Symbols 23

2.5.3 typedef asymbol 24
2.5.4 Symbol Handling Functions 26
2.5.4.1 get_symtab_upper_bound 27
2.5.4.2 bfd_canonicalize_symtab 27
2.5.4.3 bfd_set_symtab 27
2.5.4.4 bfd_print_symbol_vandf 27
2.5.4.5 bfd_make_empty_symbol 28
2.5.4.6 bfd_decode_symclass 28
2.6 Archives 28
2.6.0.1 bfd_get_next_mapent 29
2.6.0.2 bfd_set_archive_head 29
2.6.0.3 bfd_get_elt_at_index 29
2.6.0.4 bfd_openr_next_archived_file 30
2.7 File Formats 30
2.7.0.1 bfd_check_format 30
2.7.0.2 bfd_set_format 31
2.7.0.3 bfd_format_string 31
2.8 Relocations 32
2.8.1 typedef arelent 32
2.8.1.1 reloc_howto_type 35
2.8.1.2 the HOWTO macro 37
2.8.1.3 reloc_chain 37
2.8.1.4 bfd_perform_relocation 38
2.9 The howto manager 38
2.9.0.1 bfd_reloc_code_type 38
2.10 bfd_reloc_type_lookup 39
2.10.0.1 bfd_default_reloc_type_lookup 39
2.10.0.2 bfd_generic_relax_section 40
2.10.0.3 bfd_generic_get_relocated_section_contents 40
2.11 Core files 40
2.11.0.1 bfd_core_file_failing_command 40
2.11.0.2 bfd_core_file_failing_signal 41
2.11.0.3 core_file_matches_executable_p 41
2.12 Targets 41
2.12.1 bfd_target 42
2.12.1.1 bfd_find_target 46
2.12.1.2 bfd_target_list 46
2.13 Architectures 47
2.13.1 bfd_architecture 47
2.13.2 bfd_arch_info 48
2.13.2.1 bfd_printable_name 49
2.13.2.2 bfd_scan_arch 49

2.13.2.3 bfd_arch_get_compatible 49
2.13.2.4 bfd_default_arch_struct 50
2.13.2.5 bfd_set_arch_info 50
2.13.2.6 bfd_default_set_arch_mach 50
2.13.2.7 bfd_get_arch 50
2.13.2.8 bfd_get_mach 51
2.13.2.9 bfd_arch_bits_per_byte 51
2.13.2.10 bfd_arch_bits_per_address 51
2.13.2.11 bfd_arch_init 51
2.13.2.12 bfd_arch_linkin 52
2.13.2.13 bfd_default_compatible 52
2.13.2.14 bfd_default_scan 52
2.13.2.15 bfd_get_arch_info 52
2.13.2.16 bfd_lookup_arch 52
2.13.2.17 bfd_printable_arch_mach 53
2.14 Opening and Closing BFDs 53
2.14.0.1 bfd_openr 53
2.14.0.2 bfd_fdopenr 53
2.14.0.3 bfd_openw 54
2.14.0.4 bfd_close 54
2.14.0.5 bfd_close_all_done 54
2.14.0.6 bfd_create 55
2.14.0.7 bfd_alloc_by_size_t 55
2.15 Constructors 55
2.15.0.1 bfd_constructor_entry 56
2.16 libbfd 57
2.16.0.1 bfd_xmalloc 57
2.16.0.2 bfd_write_bigendian_4byte_int 57
2.16.0.3 bfd_put_size 57
2.16.0.4 bfd_get_size 57
2.16.0.5 bfd_h_put_size 58
2.16.0.6 bfd_h_get_size 58
2.16.0.7 bfd_log2 59
2.17 File Caching 59
2.17.0.1 BFD_CACHE_MAX_OPEN macro 59
2.17.0.2 bfd_last_cache 59
2.17.0.3 bfd_cache_lookup 60
2.17.0.4 bfd_cache_init 60
2.17.0.5 bfd_cache_close 60
2.17.0.6 bfd_open_file 60
2.17.0.7 bfd_cache_lookup_worker 61

3 BFD back end 62

3.1 a.out backends 62
3.1.1 relocations 63
3.1.2 Internal Entry Points 63
3.1.2.1 aout_<size>_swap_exec_header_in 63
3.1.2.2 aout_<size>_swap_exec_header_out 64
3.1.2.3 aout_<size>_some_aout_object_p 64
3.1.2.4 aout_<size>_mkobject 64
3.1.2.5 aout_<size>_machine_type 64
3.1.2.6 aout_<size>_set_arch_mach 65
3.1.2.7 aout_<size>new_section_hook 65
3.2 coff backends 65
3.2.1 Porting To A New Version of Coff 65
3.2.2 How The Coff Backend Works 66
3.2.2.1 Bit Twiddling 66
3.2.2.2 Symbol Reading 67
3.2.2.3 Symbol Writing 67
3.2.2.4 coff_symbol_type 68
3.2.2.5 Writing Relocations 69
3.2.2.6 Reading Linenumbers 69
3.2.2.7 Reading Relocations 70

Index 71