libbfd

The Binary File Descriptor Library

First Edition—BFD version < 3.0
April 1991

Steve Chamberlain
Cygnus Support

Cygnus Support
sac@cygnus.com
BFD, Revision: 1.28
TEXinfo 2.137/K

1 Introduction

BFD is a package which allows applications to use the same routines to operate on object files whatever the object file format. A new 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 back ends (one for each object file format).

- 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 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 To Use BFD

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 '`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 with '`bfd_`' to reduce namespace pollution.

For example, this sequence does 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 (see Section 2.6 [Sections], page 16),
- a set of relocations (see Section 2.10 [Relocations], page 36), and
- some symbol information (see Section 2.7 [Symbols], page 26).

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 2 Can Do

When an object file is opened, BFD subroutines automatically determine the format of the input object file. They then build a descriptor in memory with pointers to routines that will be used to access elements of the object file's data structures.

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 a memory pointer to the routine from the relevant BFD back end 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 to take the newly created symbol table and convert it into the chosen output format.

1.3.1 Information Loss

Information can be lost during output. The output formats supported by BFD do not provide identical facilities, and information which can 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 (e.g., `a.out`) or has sections without names (e.g., 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, there is no information lost 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 The BFD canonical object-file format

The greatest potential for loss of information occurs when there is the least overlap between the information provided by the source format, that stored by the canonical format, and that needed by the destination format. A brief description of the canonical form may help you understand which kinds of data you can count on preserving across conversions.

files Information stored on a per-file basis includes target machine architecture, particular implementation format type, a demand pageable bit, and a write protected bit. 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 used with one another.

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

symbols Each symbol contains a pointer to the information for 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, it relocates all symbols to make them relative to the base of the section where they were defined. Doing this ensures that each symbol points to its containing section. Each symbol also has a varying amount of hidden private data for the BFD back end. Since the symbol points to the original file, the private data format for that symbol is accessible. `ld` 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 by passing messages through the relocation

type descriptor and the symbol pointer. Therefore, relocations can be performed on output data using a relocation method that is 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 68k 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 finding out 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 has type `bfd`; objects of this type are the cornerstone of any application using BFD. Using BFD consists of making references though the BFD and to data in the BFD. Here is the structure that defines the type `bfd`. It contains the major data about the file and 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.             */
    const 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.  However, if the
       BFD_IN_MEMORY flag is set, then iostream is actually a pointer
       to a bfd_in_memory struct.  */
    PTR iostream;

    /* Is the file descriptor being cached?  That is, can it be closed as
       needed, and re-opened when accessed later?  */

    boolean cacheable;

    /* Marks whether there was a default target specified when the
       BFD was opened. This is used to select which matching algorithm
       to use to choose 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: (``once'' means at least once) */

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. (object, core, etc.) */

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
   from 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 (with symcount entries) */
struct symbol_cache_entry  **outsymbols;

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

/* Stuff only useful for archives:*/
PTR arelt_data;
struct _bfd *my_archive;     /* The containing archive BFD.  */
struct _bfd *next;           /* The next BFD in the archive.  */
struct _bfd *archive_head;   /* The first BFD in the archive.  */
boolean has_armap;

/* A chain of BFD structures involved in a link.  */
struct _bfd *link_next;

/* A field used by _bfd_generic_link_add_archive_symbols.  This will
   be used only for archive elements.  */
int archive_pass;

/* 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 pe_tdata *pe_obj_data;
  struct xcoff_tdata *xcoff_obj_data;
  struct ecoff_tdata *ecoff_obj_data;
  struct ieee_data_struct *ieee_data;
  struct ieee_ar_data_struct *ieee_ar_data;
  struct srec_data_struct *srec_data;
  struct ihex_data_struct *ihex_data;
  struct tekhex_data_struct *tekhex_data;
  struct elf_obj_tdata *elf_obj_data;
  struct nlm_obj_tdata *nlm_obj_data;
  struct bout_data_struct *bout_data;
  struct sun_core_struct *sun_core_data;
```

```
        struct trad_core_struct *trad_core_data;
        struct som_data_struct *som_data;
        struct hpux_core_struct *hpux_core_data;
        struct hppabsd_core_struct *hppabsd_core_data;
        struct sgi_core_struct *sgi_core_data;
        struct lynx_core_struct *lynx_core_data;
        struct osf_core_struct *osf_core_data;
        struct cisco_core_struct *cisco_core_data;
        struct amiga_data_struct *amiga_data;
        struct versados_data_struct *versados_data;
        struct netbsd_core_struct *netbsd_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;
 };
```

2.2 Error reporting

Most BFD functions return nonzero on success (check their individual documentation for precise semantics). On an error, they call `bfd_set_error` to set an error condition that callers can check by calling `bfd_get_error`. If that returns `bfd_error_system_call`, then check `errno`. The easiest way to report a BFD error to the user is to use `bfd_perror`.

2.2.1 Type `bfd_error_type`

The values returned by `bfd_get_error` are defined by the enumerated type `bfd_error_type`.

.

```
 typedef enum bfd_error
 {
   bfd_error_no_error = 0,
   bfd_error_system_call,
   bfd_error_invalid_target,
   bfd_error_wrong_format,
   bfd_error_invalid_operation,
   bfd_error_no_memory,
   bfd_error_no_symbols,
   bfd_error_no_armap,
   bfd_error_no_more_archived_files,
   bfd_error_malformed_archive,
   bfd_error_file_not_recognized,
```

```
  bfd_error_file_ambiguously_recognized,
  bfd_error_no_contents,
  bfd_error_nonrepresentable_section,
  bfd_error_no_debug_section,
  bfd_error_bad_value,
  bfd_error_file_truncated,
  bfd_error_file_too_big,
  bfd_error_invalid_error_code
} bfd_error_type;
```

2.2.1.1 `bfd_get_error`

Synopsis

```
bfd_error_type bfd_get_error (void);
```

Description

Return the current BFD error condition.

2.2.1.2 `bfd_set_error`

Synopsis

```
void bfd_set_error (bfd_error_type error_tag);
```

Description

Set the BFD error condition to be *error_tag*.

2.2.1.3 `bfd_errmsg`

Synopsis

```
CONST char *bfd_errmsg (bfd_error_type error_tag);
```

Description

Return a string describing the error *error_tag*, or the system error if *error_tag* is `bfd_error_system_call`.

2.2.1.4 `bfd_perror`

Synopsis

```
void bfd_perror (CONST char *message);
```

Description

Print to the standard error stream a string describing the last BFD error that occurred, or the last system error if the last BFD error was a system call failure. If *message* is non-NULL and non-empty, the error string printed is preceded by *message*, a colon, and a space. It is followed by a newline.

2.2.2 BFD error handler

Some BFD functions want to print messages describing the problem. They call a BFD error handler function. This function may be overriden by the program. The BFD error handler acts like printf.
.

```
typedef void (*bfd_error_handler_type) PARAMS ((const char *, ...));
```

2.2.2.1 `bfd_set_error_handler`

Synopsis

```
bfd_error_handler_type bfd_set_error_handler (bfd_error_handler_type);
```

Description

Set the BFD error handler function. Returns the previous function.

2.2.2.2 `bfd_set_error_program_name`

Synopsis

```
void bfd_set_error_program_name (const char *);
```

Description

Set the program name to use when printing a BFD error. This is printed before the error message followed by a colon and space. The string must not be changed after it is passed to this function.

2.3 Symbols

2.3.0.1 `bfd_get_reloc_upper_bound`

Synopsis

```
long bfd_get_reloc_upper_bound(bfd *abfd, asection *sect);
```

Description

Return the number of bytes required to store the relocation information associated with section *sect* attached to bfd *abfd*. If an error occurs, return -1.

2.3.0.2 `bfd_canonicalize_reloc`

Synopsis

```
long bfd_canonicalize_reloc
   (bfd *abfd,
    asection *sec,
    arelent **loc,
```

```
        asymbol **syms);
```

Description
Call the back end associated with the open BFD *abfd* and translate the external form of the relocation information attached to *sec* into the internal canonical form. Place the table into memory at *loc*, which has been preallocated, usually by a call to `bfd_get_reloc_upper_bound`. Returns the number of relocs, or -1 on error. The *syms* table is also needed for horrible internal magic reasons.

2.3.0.3 `bfd_set_reloc`

Synopsis

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

Description
Set the relocation pointer and count within section *sec* to the values *rel* and *count*. The argument *abfd* is ignored.

2.3.0.4 `bfd_set_file_flags`

Synopsis

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

Description
Set the flag word in the BFD *abfd* to the value *flags*. Possible errors are:

- `bfd_error_wrong_format` - The target bfd was not of object format.
- `bfd_error_invalid_operation` - The target bfd was open for reading.
- `bfd_error_invalid_operation` - The flag word contained a bit which was not applicable to the type of file. E.g., an attempt was made to set the `D_PAGED` bit on a BFD format which does not support demand paging.

2.3.0.5 `bfd_set_start_address`

Synopsis

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

Description
Make *vma* the entry point of output BFD *abfd*.
Returns
Returns `true` on success, `false` otherwise.

2.3.0.6 bfd_get_mtime

Synopsis
```
long bfd_get_mtime(bfd *abfd);
```
Description
Return the file modification time (as read from the file system, or from the archive header for archive members).

2.3.0.7 bfd_get_size

Synopsis
```
long bfd_get_size(bfd *abfd);
```
Description
Return the file size (as read from file system) for the file associated with BFD *abfd*. The initial motivation for, and use of, this routine is not so we can get the exact size of the object the BFD applies to, since that might not be generally possible (archive members for example). It would be ideal if someone could eventually modify it so that such results were guaranteed. Instead, we want to ask questions like "is this NNN byte sized object I'm about to try read from file offset YYY reasonable?" As as example of where we might do this, some object formats use string tables for which the first `sizeof(long)` bytes of the table contain the size of the table itself, including the size bytes. If an application tries to read what it thinks is one of these string tables, without some way to validate the size, and for some reason the size is wrong (byte swapping error, wrong location for the string table, etc.), the only clue is likely to be a read error when it tries to read the table, or a "virtual memory exhausted" error when it tries to allocate 15 bazillon bytes of space for the 15 bazillon byte table it is about to read. This function at least allows us to answer the quesion, "is the size reasonable?".

2.3.0.8 bfd_get_gp_size

Synopsis
```
int bfd_get_gp_size(bfd *abfd);
```
Description
Return the maximum size of objects to be optimized using the GP register under MIPS ECOFF. This is typically set by the `-G` argument to the compiler, assembler or linker.

2.3.0.9 bfd_set_gp_size

Synopsis
```
void bfd_set_gp_size(bfd *abfd, int i);
```
Description
Set the maximum size of objects to be optimized using the GP register under ECOFF or MIPS ELF. This is typically set by the `-G` argument to the compiler, assembler or linker.

2.3.0.10 `bfd_scan_vma`

Synopsis

```
bfd_vma bfd_scan_vma(CONST char *string, CONST char **end, int base);
```

Description

Convert, like `strtoul`, a numerical expression *string* into a `bfd_vma` integer, and return that integer. (Though without as many bells and whistles as `strtoul`.) The expression is assumed to be unsigned (i.e., positive). If given a *base*, it is used as the base for conversion. A base of 0 causes the function to interpret the string in hex if a leading "0x" or "0X" is found, otherwise in octal if a leading zero is found, otherwise in decimal. Overflow is not detected.

2.3.0.11 `bfd_copy_private_bfd_data`

Synopsis

```
boolean bfd_copy_private_bfd_data(bfd *ibfd, bfd *obfd);
```

Description

Copy private BFD information from the BFD *ibfd* to the the BFD *obfd*. Return `true` on success, `false` on error. Possible error returns are:

- `bfd_error_no_memory` - Not enough memory exists to create private data for *obfd*.

```
#define bfd_copy_private_bfd_data(ibfd, obfd) \
     BFD_SEND (ibfd, _bfd_copy_private_bfd_data, \
(ibfd, obfd))
```

2.3.0.12 `bfd_merge_private_bfd_data`

Synopsis

```
boolean bfd_merge_private_bfd_data(bfd *ibfd, bfd *obfd);
```

Description

Merge private BFD information from the BFD *ibfd* to the the output file BFD *obfd* when linking. Return `true` on success, `false` on error. Possible error returns are:

- `bfd_error_no_memory` - Not enough memory exists to create private data for *obfd*.

```
#define bfd_merge_private_bfd_data(ibfd, obfd) \
     BFD_SEND (ibfd, _bfd_merge_private_bfd_data, \
(ibfd, obfd))
```

2.3.0.13 `bfd_set_private_flags`

Synopsis

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

Description

Set private BFD flag information in the BFD *abfd*. Return `true` on success, `false` on error. Possible error returns are:

- `bfd_error_no_memory` - Not enough memory exists to create private data for *obfd*.

```
#define bfd_set_private_flags(abfd, flags) \
     BFD_SEND (abfd, _bfd_set_private_flags, \
(abfd, flags))
```

2.3.0.14 `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))

        /* Do these three do anything useful at all, for any back end?  */
#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_update_armap_timestamp(abfd) \
        BFD_SEND (abfd, _bfd_update_armap_timestamp, (abfd))

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

#define bfd_relax_section(abfd, section, link_info, again) \
       BFD_SEND (abfd, _bfd_relax_section, (abfd, section, link_info, again))

#define bfd_link_hash_table_create(abfd) \
BFD_SEND (abfd, _bfd_link_hash_table_create, (abfd))

#define bfd_link_add_symbols(abfd, info) \
BFD_SEND (abfd, _bfd_link_add_symbols, (abfd, info))

#define bfd_final_link(abfd, info) \
BFD_SEND (abfd, _bfd_final_link, (abfd, info))
```

```
#define bfd_free_cached_info(abfd) \
       BFD_SEND (abfd, _bfd_free_cached_info, (abfd))

#define bfd_get_dynamic_symtab_upper_bound(abfd) \
BFD_SEND (abfd, _bfd_get_dynamic_symtab_upper_bound, (abfd))

#define bfd_print_private_bfd_data(abfd, file)\
BFD_SEND (abfd, _bfd_print_private_bfd_data, (abfd, file))

#define bfd_canonicalize_dynamic_symtab(abfd, asymbols) \
BFD_SEND (abfd, _bfd_canonicalize_dynamic_symtab, (abfd, asymbols))

#define bfd_get_dynamic_reloc_upper_bound(abfd) \
BFD_SEND (abfd, _bfd_get_dynamic_reloc_upper_bound, (abfd))

#define bfd_canonicalize_dynamic_reloc(abfd, arels, asyms) \
BFD_SEND (abfd, _bfd_canonicalize_dynamic_reloc, (abfd, arels, asyms))

extern bfd_byte *bfd_get_relocated_section_contents
PARAMS ((bfd *, struct bfd_link_info *,
  struct bfd_link_order *, bfd_byte *,
  boolean, asymbol **));
```

2.4 Memory usage

BFD keeps all of 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 BFD for the closing file is thrown away.

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

The general rule is to not 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.

2.5 Initialization

These are the functions that handle initializing a BFD.

2.5.0.1 `bfd_init`

Synopsis

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

Description

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

2.6 Sections

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

2.6.1 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`. Names need not be unique; for example a COFF file may have several sections named `.data`. 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 an extra section `COMMON` for each input file's BFD to hold information about common storage. The raw data is not necessarily read in when 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.

2.6.2 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`. Any program that creates or combines sections (e.g., the assembler and linker) must use the `asection` fields `output_section` and `output_offset` to indicate the file sections to which each section must be written. (If the section is being created from scratch, `output_section` should probably point to the section itself and `output_offset` should probably be zero.) The data to be written comes from input sections attached (via `output_section` pointers) 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. E.g., to create a section

"O", starting at 0x100, 0x123 long, containing two subsections, "A" at offset 0x0 (i.e., at vma 0x100) and "B" at offset 0x20 (i.e., at vma 0x120) the `asection` 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  --------|
```

2.6.3 Link orders

The data within a section is stored in a *link_order*. These are much like the fixups in `gas`. The link_order abstraction allows a section to grow and shrink within itself. A link_order knows how big it is, and which is the next link_order and where the raw data for it is; it also points to a list of relocations which apply to it. The link_order is used by the linker to perform relaxing on final code. The compiler 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 link_order by link_order basis.

2.6.4 typedef asection

Here is the section structure:

.

```
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
           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 loading.
                This is 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 is clear for a .bss section. */
     #define SEC_LOAD       0x002

             /* The section contains data still to be relocated, so there is
                some relocation information too. */
     #define SEC_RELOC      0x004

     #if 0   /* Obsolete ? */
     #define SEC_BALIGN     0x008
     #endif

             /* 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 (e.g., __CTOR_LIST__), attaches
                the symbol to it, and builds a relocation. To build the lists
                of constructors, all the linker has to do is catenate all the
                sections called __CTOR_LIST__ and relocate 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 text, data, or bss section(?). */
#define SEC_CONSTRUCTOR_TEXT 0x1100
#define SEC_CONSTRUCTOR_DATA 0x2100
#define SEC_CONSTRUCTOR_BSS  0x3100

        /* The section has contents - a data 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 to not output the section
           even if it has information which would normally be written. */
#define SEC_NEVER_LOAD 0x400

        /* The section is a COFF shared library section.  This flag is
           only for the linker.  If this type of section appears in
           the input file, the linker must copy it to the output file
           without changing the vma or size.  FIXME: Although this
           was originally intended to be general, it really is COFF
           specific (and the flag was renamed to indicate this).  It
           might be cleaner to have some more general mechanism to
           allow the back end to control what the linker does with
           sections. */
#define SEC_COFF_SHARED_LIBRARY 0x800

        /* The section is a common section (symbols may be defined
           multiple times, the value of a symbol is the amount of
           space it requires, and the largest symbol value is the one
           used).  Most targets have exactly one of these (which we
    translate to bfd_com_section_ptr), but ECOFF has two. */
#define SEC_IS_COMMON 0x8000

        /* The section contains only debugging information.  For
           example, this is set for ELF .debug and .stab sections.
           strip tests this flag to see if a section can be
           discarded. */
#define SEC_DEBUGGING 0x10000

        /* The contents of this section are held in memory pointed to
           by the contents field.  This is checked by
           bfd_get_section_contents, and the data is retrieved from
           memory if appropriate.  */
#define SEC_IN_MEMORY 0x20000

        /* The contents of this section are to be excluded by the
    linker for executable and shared objects unless those
    objects are to be further relocated.  */
```

```
#define SEC_EXCLUDE 0x40000

/* The contents of this section are to be sorted by the
   based on the address specified in the associated symbol
   table.  */
#define SEC_SORT_ENTRIES 0x80000

/*  End of section flags.  */

       /*  The virtual memory address of the section - where it will be
           at run time.  The symbols are relocated against this.  The
    user_set_vma flag is maintained by bfd; if it's not set, the
    backend can assign addresses (for example, in a.out, where
    the default address for .data is dependent on the specific
    target and various flags).  */

   bfd_vma vma;
   boolean user_set_vma;

       /*  The load address of the section - where it would be in a
           rom image; really only used for writing section header
    information. */

   bfd_vma lma;

        /* The size of the section in bytes, as it will be output.
           contains a value even if the section has no contents (e.g., the
           size of .bss). This will be filled in after relocation */

   bfd_size_type _cooked_size;

        /* The original size on disk of the section, in bytes.  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. E.g., 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 of 2 -
        e.g., 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;

     /* If the SEC_IN_MEMORY flag is set, this points to the actual
        contents.  */
unsigned char *contents;

     /* 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_link_order *link_order_head;
   struct bfd_link_order *link_order_tail;
} asection ;

    /* These sections are global, and are managed by BFD.  The application
       and target back end are not permitted to change the values in
these sections.  New code should use the section_ptr macros rather
       than referring directly to the const sections.  The const sections
       may eventually vanish.  */
#define BFD_ABS_SECTION_NAME "*ABS*"
#define BFD_UND_SECTION_NAME "*UND*"
#define BFD_COM_SECTION_NAME "*COM*"
#define BFD_IND_SECTION_NAME "*IND*"

    /* the absolute section */
extern const asection bfd_abs_section;
#define bfd_abs_section_ptr ((asection *) &bfd_abs_section)
#define bfd_is_abs_section(sec) ((sec) == bfd_abs_section_ptr)
    /* Pointer to the undefined section */
extern const asection bfd_und_section;
#define bfd_und_section_ptr ((asection *) &bfd_und_section)
```

```
#define bfd_is_und_section(sec) ((sec) == bfd_und_section_ptr)
    /* Pointer to the common section */
extern const asection bfd_com_section;
#define bfd_com_section_ptr ((asection *) &bfd_com_section)
    /* Pointer to the indirect section */
extern const asection bfd_ind_section;
#define bfd_ind_section_ptr ((asection *) &bfd_ind_section)
#define bfd_is_ind_section(sec) ((sec) == bfd_ind_section_ptr)

extern const struct symbol_cache_entry * const bfd_abs_symbol;
extern const struct symbol_cache_entry * const bfd_com_symbol;
extern const struct symbol_cache_entry * const bfd_und_symbol;
extern const struct symbol_cache_entry * const bfd_ind_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))
```

2.6.5 Section prototypes

These are the functions exported by the section handling part of BFD.

2.6.5.1 `bfd_get_section_by_name`

Synopsis

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

Description

Run through *abfd* and return the one of the `asection`s whose name matches *name*, otherwise `NULL`. See Section 2.6 [Sections], page 16, for more information. This should only be used in special cases; the normal way to process all sections of a given name is to use `bfd_map_over_sections` and `strcmp` on the name (or better yet, base it on the section flags or something else) for each section.

2.6.5.2 `bfd_make_section_old_way`

Synopsis

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

Description

Create a new empty section called *name* and attach it to the end of the chain of sections for the BFD *abfd*. 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 it was rewritten.... Possible errors are:

- `bfd_error_invalid_operation` - If output has already started for this BFD.
- `bfd_error_no_memory` - If obstack alloc fails.

2.6.5.3 `bfd_make_section_anyway`

Synopsis

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

Description

Create a new empty section called *name* and attach it to the end of the chain of sections for *abfd*. Create a new section even if there is already a section with that name. Return `NULL` and set `bfd_error` on error; possible errors are:

- `bfd_error_invalid_operation` - If output has already started for *abfd*.
- `bfd_error_no_memory` - If obstack alloc fails.

2.6.5.4 `bfd_make_section`

Synopsis

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

Description

Like `bfd_make_section_anyway`, but return `NULL` (without calling bfd_set_error ()) without changing the section chain if there is already a section named *name*. If there is an error, return `NULL` and set `bfd_error`.

2.6.5.5 `bfd_set_section_flags`

Synopsis

```
boolean bfd_set_section_flags(bfd *abfd, asection *sec, flagword flags);
```

Description

Set the attributes of the section *sec* in the BFD *abfd* to the value *flags*. Return `true` on success, `false` on error. Possible error returns are:

- `bfd_error_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.

2.6.5.6 `bfd_map_over_sections`

Synopsis

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

Description

Call 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, ...)
```

2.6.5.7 `bfd_set_section_size`

Synopsis

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

Description

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

- `bfd_error_invalid_operation` - Writing has started to the BFD, so setting the size is invalid.

2.6.5.8 `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:

- `bfd_error_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`.

2.6.5.9 `bfd_get_section_contents`

Synopsis

```
boolean bfd_get_section_contents
```

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

Description
Read 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 constructor with the `SEC_CONSTRUCTOR` flag set are requested or if the section does not have the `SEC_HAS_CONTENTS` flag set, then the *location* is filled with zeroes. If no errors occur, `true` is returned, else `false`.

2.6.5.10 `bfd_copy_private_section_data`

Synopsis

```
boolean bfd_copy_private_section_data(bfd *ibfd, asection *isec, bfd *obfd, asection *osec);
```

Description
Copy private section information from *isec* in the BFD *ibfd* to the section *osec* in the BFD *obfd*. Return `true` on success, `false` on error. Possible error returns are:

- `bfd_error_no_memory` - Not enough memory exists to create private data for *osec*.

```
#define bfd_copy_private_section_data(ibfd, isection, obfd, osection) \
     BFD_SEND (ibfd, _bfd_copy_private_section_data, \
(ibfd, isection, obfd, osection))
```

2.7 Symbols

BFD tries 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 information passed to applications, some targets keep some information "behind the scenes" 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 were read, but were 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 `asymbol`s. This allows applications like the linker to output a symbol as it was read, since the "behind the scenes" information will be still available.

2.7.1 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 application which reads the symbol table:

```
       long storage_needed;
       asymbol **symbol_table;
       long number_of_symbols;
       long i;

       storage_needed = bfd_get_symtab_upper_bound (abfd);

              if (storage_needed < 0)
                FAIL

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

              if (number_of_symbols < 0)
                FAIL

       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; it is freed when the BFD is closed.

2.7.2 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 BFD output 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`. Here is 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.

2.7.3 Mini Symbols

Mini symbols provide read-only access to the symbol table. They use less memory space, but require more time to access. They can be useful for tools like nm or objdump, which may have to handle symbol tables of extremely large executables. The `bfd_read_minisymbols` function will read the symbols into memory in an internal form. It will return a `void *` pointer to a block of memory, a symbol count, and the size of each symbol. The pointer is allocated using `malloc`, and should be freed by the caller when it is no longer needed. The function `bfd_minisymbol_to_symbol` will take a pointer to a minisymbol, and a pointer to a structure returned by `bfd_make_empty_symbol`, and return a `asymbol` structure. The return value may or may not be the same as the value from `bfd_make_empty_symbol` which was passed in.

2.7.4 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
       information (invisible to the application writer) is carried
   with the symbol.

   This field is *almost* redundant, since you can use section->owner
   instead, except that some symbols point to the global sections
   bfd_{abs,com,und}_section.  This could be fixed by making
   these globals be per-bfd (or per-target-flavor).  FIXME. */
```

```
  struct _bfd *the_bfd; /* Use bfd_asymbol_bfd(sym) to access this field. */

/* 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.  This really should be a union of a
          numeric value with a pointer, since some flags indicate that
          a pointer to another symbol is stored here.  */
  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

/* The symbol has global scope and is exported. The value is
   the offset into the section of the data. */
#define BSF_EXPORT BSF_GLOBAL /* no real difference */

/* A normal C symbol would be one of:
   BSF_LOCAL, BSF_FORT_COMM,  BSF_UNDEFINED or
   BSF_GLOBAL */

/* The symbol is a debugging record. The value has an arbitary
   meaning. */
#define BSF_DEBUGGING 0x08

/* The symbol denotes a function entry point.  Used in ELF,
   perhaps others someday.  */
#define BSF_FUNCTION    0x10

/* Used by the linker. */
#define BSF_KEEP        0x20
#define BSF_KEEP_G      0x40

/* A weak global symbol, overridable without warnings by
   a regular global symbol of the same name.  */
#define BSF_WEAK        0x80

       /* This symbol was created to point to a section, e.g. ELF's
```

```
   STT_SECTION symbols.  */
#define BSF_SECTION_SYM 0x100

/* The symbol used to be a common symbol, but now it is
   allocated. */
#define BSF_OLD_COMMON  0x200

/* 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    0x400

/* Signal that the symbol is the label of constructor section. */
#define BSF_CONSTRUCTOR   0x800

/* Signal that the symbol is a warning symbol.  The name is a
   warning.  The name of the next symbol is the one to warn about;
   if a reference is made to a symbol with the same name as the next
   symbol, a warning is issued by the linker. */
#define BSF_WARNING       0x1000

/* Signal that the symbol is indirect.  This symbol is an indirect
   pointer to the symbol with the same name as the next symbol. */
#define BSF_INDIRECT      0x2000

/* BSF_FILE marks symbols that contain a file name.  This is used
   for ELF STT_FILE symbols.  */
#define BSF_FILE          0x4000

/* Symbol is from dynamic linking information.  */
#define BSF_DYNAMIC    0x8000

       /* The symbol denotes a data object.  Used in ELF, and perhaps
          others someday.  */
#define BSF_OBJECT    0x10000

  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.  */
  union
    {
      PTR p;
      bfd_vma i;
    } udata;

} asymbol;
```

2.7.5 Symbol handling functions

2.7.5.1 `bfd_get_symtab_upper_bound`

Description
Return the number of bytes required to store a vector of pointers to `asymbols` for all the symbols in the BFD *abfd*, including a terminal NULL pointer. If there are no symbols in the BFD, then return 0. If an error occurs, return -1.

```
#define bfd_get_symtab_upper_bound(abfd) \
     BFD_SEND (abfd, _bfd_get_symtab_upper_bound, (abfd))
```

2.7.5.2 `bfd_is_local_label`

Synopsis

```
boolean bfd_is_local_label(bfd *abfd, asymbol *sym);
```

Description
Return true if the given symbol *sym* in the BFD *abfd* is a compiler generated local label, else return false.

```
#define bfd_is_local_label(abfd, sym) \
     BFD_SEND (abfd, _bfd_is_local_label,(abfd, sym))
```

2.7.5.3 `bfd_canonicalize_symtab`

Description
Read the symbols from the BFD *abfd*, and fills in the vector *location* with pointers to the symbols and a trailing NULL. Return the actual number of symbol pointers, not including the NULL.

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

2.7.5.4 `bfd_set_symtab`

Synopsis

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

Description

Arrange that when the output BFD *abfd* is closed, the table *location* of *count* pointers to symbols will be written.

2.7.5.5 `bfd_print_symbol_vandf`

Synopsis

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

Description

Print the value and flags of the *symbol* supplied to the stream *file*.

2.7.5.6 `bfd_make_empty_symbol`

Description

Create a new `asymbol` structure for the BFD *abfd* and return a pointer to it. This routine is necessary because 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))
```

2.7.5.7 `bfd_make_debug_symbol`

Description

Create a new `asymbol` structure for the BFD *abfd*, to be used as a debugging symbol. Further details of its use have yet to be worked out.

```
#define bfd_make_debug_symbol(abfd,ptr,size) \
        BFD_SEND (abfd, _bfd_make_debug_symbol, (abfd, ptr, size))
```

2.7.5.8 `bfd_decode_symclass`

Description

Return a character corresponding to the symbol class of *symbol*, or '?' for an unknown class.

Synopsis

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

2.7.5.9 `bfd_symbol_info`

Description
Fill in the basic info about symbol that nm needs. Additional info may be added by the back-ends after calling this function.
Synopsis
```
void bfd_symbol_info(asymbol *symbol, symbol_info *ret);
```

2.7.5.10 `bfd_copy_private_symbol_data`

Synopsis
```
boolean bfd_copy_private_symbol_data(bfd *ibfd, asymbol *isym, bfd *obfd, asymbol *osym);
```
Description
Copy private symbol information from *isym* in the BFD *ibfd* to the symbol *osym* in the BFD *obfd*. Return `true` on success, `false` on error. Possible error returns are:

- `bfd_error_no_memory` - Not enough memory exists to create private data for *osec*.

```
#define bfd_copy_private_symbol_data(ibfd, isymbol, obfd, osymbol) \
     BFD_SEND (ibfd, _bfd_copy_private_symbol_data, \
(ibfd, isymbol, obfd, osymbol))
```

2.8 Archives

Description
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 that are considered its contents. These BFDs can be manipulated 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; they will be handled correctly when the archive is closed. Use `bfd_openr_next_archived_file` to step through the contents of an archive opened for input. You don't have to 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 `bfd_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 (e.g., 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; SunOS 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 case due to restrictions 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. Archives are supported in BFD in `archive.c`.

2.8.0.1 `bfd_get_next_mapent`

Synopsis

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

Description

Step through archive *abfd*'s symbol table (if it has one). Successively update *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've already got the last one. A `carsym` is a canonical archive symbol. The only user-visible element is its name, a null-terminated string.

2.8.0.2 `bfd_set_archive_head`

Synopsis

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

Description

Set the head of the chain of BFDs contained in the archive *output* to *new_head*.

2.8.0.3 `bfd_openr_next_archived_file`

Synopsis

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

Description

Provided a BFD, *archive*, containing an archive and NULL, open an input BFD on the first contained element and returns that. Subsequent calls 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.

2.9 File formats

A format is a BFD concept of high level file contents type. 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.

2.9.0.1 `bfd_check_format`

Synopsis

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

Description

Verify if the file attached to the BFD *abfd* is compatible with the format *format* (i.e., 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 is checked. If the target has not been set, or has been set to `default`, then all the known target backends is interrogated to determine a match. If the default target matches, it is used. If not, exactly one target must recognize the file, or an error results. The function returns `true` on success, otherwise `false` with one of the following error codes:

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

2.9.0.2 `bfd_check_format_matches`

Synopsis

```
boolean bfd_check_format_matches(bfd *abfd, bfd_format format, char ***matching);
```

Description

Like `bfd_check_format`, except when it returns false with `bfd_errno` set to `bfd_error_file_ambiguously_recognized`. In that case, if *matching* is not NULL, it will be filled in with a NULL-terminated list of the names of the formats that matched, allocated with `malloc`. Then the user may choose a format and try again. When done with the list that *matching* points to, the caller should free it.

2.9.0.3 `bfd_set_format`

Synopsis

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

Description

This function sets the file format of the BFD *abfd* to the format *format*. If the target set in the BFD does not support the format requested, the format is invalid, or the BFD is not open for writing, then an error occurs.

2.9.0.4 bfd_format_string

Synopsis

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

Description

Return a pointer to a const string `invalid`, `object`, `archive`, `core`, or `unknown`, depending upon the value of *format*.

2.10 Relocations

BFD maintains relocations in much the same way it maintains symbols: they are left alone until required, then read in en-mass and translated into an internal form. A common routine `bfd_perform_relocation` acts upon the canonical form to do the fixup. Relocations are maintained on a per section basis, while symbols are maintained on a per BFD basis. All that a back end has to do to fit the BFD interface is to create a `struct reloc_cache_entry` for each relocation in a particular section, and fill in the right bits of the structures.

2.10.1 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,

       /* Unsupported relocation size requested. */
  bfd_reloc_notsupported,

       /* Unused */
  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.  If this type is returned, the error_message argument
               to bfd_perform_relocation will be set.  */
       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 */
       bfd_size_type address;

            /* addend for relocation value */
       bfd_vma addend;

            /* Pointer to how to perform the required relocation */
       reloc_howto_type *howto;

     } arelent;
```

Description

Here is a description of each of the fields within an `arelent`:

- `sym_ptr_ptr`

The symbol table pointer points to a pointer to the symbol associated with the relocation request. It is the pointer into the table returned by the back end's `get_symtab` action. See Section 2.7 [Symbols], page 26. 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 should 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. One 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 in odd sized lumps. The designers of the a.out format chose to not use the data within the section for storing part of the offset; all the offset is kept within the reloc. Anything 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 contain a pointer to `foo`, and the offsets 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 structure which contains information on what to do with all of 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.

2.10.1.1 `enum complain_overflow`

Indicates what sort of overflow checking should be done when performing a relocation.

.

```
enum complain_overflow
{
/* Do not complain on overflow. */
  complain_overflow_dont,

/* Complain if the bitfield overflows, whether it is considered
   as signed or unsigned. */
  complain_overflow_bitfield,

/* Complain if the value overflows when considered as signed
   number. */
  complain_overflow_signed,

/* Complain if the value overflows when considered as an
   unsigned number. */
  complain_overflow_unsigned
};
```

2.10.1.2 `reloc_howto_type`

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

```
struct reloc_howto_struct
{
       /*  The type field has mainly a documetary use - the back end can
           do what it wants with it, though normally the back end's
           external idea of what a reloc number is stored
```

```
                 in this field. For example, a PC relative word relocation
                 in a coff environment has 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.  This is *not* a
     power-of-two measure.  To get the number of bytes operated
     on by a type of relocation, use bfd_get_reloc_size.  */
  int size;

       /*  The number of bits in the item to be relocated.  This is used
     when doing overflow checking.  */
  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;

 /*  The bit position of the reloc value in the destination.
     The relocated value is left shifted by this amount. */
  unsigned int bitpos;

 /* What type of overflow error should be checked for when
    relocating. */
  enum complain_overflow 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 (e.g., i960 callj
          instructions). */
  bfd_reloc_status_type (*special_function)
    PARAMS ((bfd *abfd,
     arelent *reloc_entry,
                                            struct symbol_cache_entry *symbol,
                                            PTR data,
                                            asection *input_section,
                                            bfd *output_bfd,
                                            char **error_message));

       /* 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 selects which parts of the read in data
           are to be used in the relocation sum.  E.g., 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_vma src_mask;

        /* The dst_mask selects which 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_vma 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 (e.g., sun3 a.out).
           Some formats leave the displacement part of an instruction
           empty (e.g., m88k bcs); this flag signals the fact.*/
   boolean pcrel_offset;

 };
```

2.10.1.3 The HOWTO Macro

Description
The HOWTO define is horrible and will go away.

```
#define HOWTO(C, R,S,B, P, BI, O, SF, NAME, INPLACE, MASKSRC, MASKDST, PC) \
  {(unsigned)C,R,S,B, P, BI, 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,complain_overflow_dont,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 (bfd_is_com_section (symbol->section)) { \
```

```
          relocation = 0;                         \
        }                                         \
        else {                                    \
          relocation = symbol->value;             \
        }                                         \
      }                                           \
    }
```

2.10.1.4 `bfd_get_reloc_size`

Synopsis

```
int bfd_get_reloc_size (reloc_howto_type *);
```

Description

For a reloc_howto_type that operates on a fixed number of bytes, this returns the number of bytes operated on.

2.10.1.5 `arelent_chain`

Description

How relocs are tied together in an `asection`:

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

2.10.1.6 `bfd_perform_relocation`

Synopsis

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

Description

If *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 (e.g., 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. The *error_message* argument is set to an error message if this return `bfd_reloc_dangerous`.

2.10.1.7 `bfd_install_relocation`

Synopsis

```
bfd_reloc_status_type
bfd_install_relocation
   (bfd *abfd,
    arelent *reloc_entry,
    PTR data, bfd_vma data_start,
    asection *input_section,
    char **error_message);
```

Description

This looks remarkably like `bfd_perform_relocation`, except it does not expect that the section contents have been filled in. I.e., it's suitable for use when creating, rather than applying a relocation. For now, this function should be considered reserved for the assembler.

2.11 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.

2.11.0.1 `bfd_reloc_code_type`

Description

The insides of a reloc code. The idea is that, eventually, there will be one enumerator for every type of relocation we ever do. Pass one of these values to `bfd_reloc_type_lookup`, and it'll return a howto pointer. This does mean that the application must determine the correct enumerator value; you can't get a howto pointer from a random set of attributes.
Here are the possible values for `enum bfd_reloc_code_real`:

BFD_RELOC_64
BFD_RELOC_32
BFD_RELOC_26
BFD_RELOC_16
BFD_RELOC_14
BFD_RELOC_8

Basic absolute relocations of N bits.

BFD_RELOC_64_PCREL
BFD_RELOC_32_PCREL
BFD_RELOC_24_PCREL
BFD_RELOC_16_PCREL
BFD_RELOC_12_PCREL
BFD_RELOC_8_PCREL

PC-relative relocations. Sometimes these are relative to the address of the relocation itself; sometimes they are relative to the start of the section containing the relocation. It depends on the specific target.

The 24-bit relocation is used in some Intel 960 configurations.

BFD_RELOC_32_GOT_PCREL
BFD_RELOC_16_GOT_PCREL
BFD_RELOC_8_GOT_PCREL
BFD_RELOC_32_GOTOFF
BFD_RELOC_16_GOTOFF
BFD_RELOC_LO16_GOTOFF
BFD_RELOC_HI16_GOTOFF
BFD_RELOC_HI16_S_GOTOFF
BFD_RELOC_8_GOTOFF
BFD_RELOC_32_PLT_PCREL
BFD_RELOC_24_PLT_PCREL
BFD_RELOC_16_PLT_PCREL
BFD_RELOC_8_PLT_PCREL
BFD_RELOC_32_PLTOFF
BFD_RELOC_16_PLTOFF
BFD_RELOC_LO16_PLTOFF
BFD_RELOC_HI16_PLTOFF
BFD_RELOC_HI16_S_PLTOFF
BFD_RELOC_8_PLTOFF

For ELF.

BFD_RELOC_68K_GLOB_DAT
BFD_RELOC_68K_JMP_SLOT
BFD_RELOC_68K_RELATIVE

Relocations used by 68K ELF.

BFD_RELOC_32_BASEREL
BFD_RELOC_16_BASEREL
BFD_RELOC_LO16_BASEREL
BFD_RELOC_HI16_BASEREL
BFD_RELOC_HI16_S_BASEREL
BFD_RELOC_8_BASEREL
BFD_RELOC_RVA

Linkage-table relative.

BFD_RELOC_8_FFnn

Absolute 8-bit relocation, but used to form an address like 0xFFnn.

BFD_RELOC_32_PCREL_S2
BFD_RELOC_16_PCREL_S2
BFD_RELOC_23_PCREL_S2

These PC-relative relocations are stored as word displacements – i.e., byte displacements shifted right two bits. The 30-bit word displacement (<<32_PCREL_S2>> – 32 bits, shifted 2) is used on the SPARC. (SPARC tools generally refer to this as <<WDISP30>>.) The signed 16-bit displacement is used on the MIPS, and the 23-bit displacement is used on the Alpha.

BFD_RELOC_HI22
BFD_RELOC_LO10

High 22 bits and low 10 bits of 32-bit value, placed into lower bits of the target word. These are used on the SPARC.

BFD_RELOC_GPREL16
BFD_RELOC_GPREL32

For systems that allocate a Global Pointer register, these are displacements off that register. These relocation types are handled specially, because the value the register will have is decided relatively late.

BFD_RELOC_I960_CALLJ

Reloc types used for i960/b.out.

BFD_RELOC_NONE
BFD_RELOC_SPARC_WDISP22
BFD_RELOC_SPARC22
BFD_RELOC_SPARC13
BFD_RELOC_SPARC_GOT10
BFD_RELOC_SPARC_GOT13
BFD_RELOC_SPARC_GOT22
BFD_RELOC_SPARC_PC10
BFD_RELOC_SPARC_PC22
BFD_RELOC_SPARC_WPLT30
BFD_RELOC_SPARC_COPY
BFD_RELOC_SPARC_GLOB_DAT
BFD_RELOC_SPARC_JMP_SLOT
BFD_RELOC_SPARC_RELATIVE
BFD_RELOC_SPARC_UA32

SPARC ELF relocations. There is probably some overlap with other relocation types already defined.

BFD_RELOC_SPARC_BASE13
BFD_RELOC_SPARC_BASE22

I think these are specific to SPARC a.out (e.g., Sun 4).

BFD_RELOC_SPARC_64
BFD_RELOC_SPARC_10
BFD_RELOC_SPARC_11
BFD_RELOC_SPARC_OLO10
BFD_RELOC_SPARC_HH22
BFD_RELOC_SPARC_HM10
BFD_RELOC_SPARC_LM22
BFD_RELOC_SPARC_PC_HH22
BFD_RELOC_SPARC_PC_HM10
BFD_RELOC_SPARC_PC_LM22
BFD_RELOC_SPARC_WDISP16
BFD_RELOC_SPARC_WDISP19
BFD_RELOC_SPARC_GLOB_JMP
BFD_RELOC_SPARC_7
BFD_RELOC_SPARC_6
BFD_RELOC_SPARC_5

Some relocations we're using for SPARC V9 – subject to change.

BFD_RELOC_ALPHA_GPDISP_HI16

Alpha ECOFF relocations. Some of these treat the symbol or "addend" in some special way. For GPDISP_HI16 ("gpdisp") relocations, the symbol is ignored when writing; when reading, it will be the absolute section symbol. The addend is the displacement in bytes of the "lda" instruction from the "ldah" instruction (which is at the address of this reloc).

BFD_RELOC_ALPHA_GPDISP_LO16

For GPDISP_LO16 ("ignore") relocations, the symbol is handled as with GPDISP_HI16 relocs. The addend is ignored when writing the relocations out, and is filled in with the file's GP value on reading, for convenience.

BFD_RELOC_ALPHA_LITERAL
BFD_RELOC_ALPHA_LITUSE

The Alpha LITERAL/LITUSE relocs are produced by a symbol reference; the assembler turns it into a LDQ instruction to load the address of the symbol, and then fills in a register in the real instruction.

The LITERAL reloc, at the LDQ instruction, refers to the .lita section symbol. The addend is ignored when writing, but is filled in with the file's GP value on reading, for convenience, as with the GPDISP_LO16 reloc.

The LITUSE reloc, on the instruction using the loaded address, gives information to the linker that it might be able to use to optimize away some literal section references. The symbol is ignored (read as the absolute section symbol), and the "addend" indicates the type of instruction using the register: 1 - "memory" fmt insn 2 - byte-manipulation (byte offset reg) 3 - jsr (target of branch)

The GNU linker currently doesn't do any of this optimizing.

BFD_RELOC_ALPHA_HINT

The HINT relocation indicates a value that should be filled into the "hint" field of a jmp/jsr/ret instruction, for possible branch- prediction logic which may be provided on some processors.

BFD_RELOC_MIPS_JMP
Bits 27..2 of the relocation address shifted right 2 bits; simple reloc otherwise.

BFD_RELOC_HI16
High 16 bits of 32-bit value; simple reloc.

BFD_RELOC_HI16_S
High 16 bits of 32-bit value but the low 16 bits will be sign extended and added to form the final result. If the low 16 bits form a negative number, we need to add one to the high value to compensate for the borrow when the low bits are added.

BFD_RELOC_LO16
Low 16 bits.

BFD_RELOC_PCREL_HI16_S
Like BFD_RELOC_HI16_S, but PC relative.

BFD_RELOC_PCREL_LO16
Like BFD_RELOC_LO16, but PC relative.

BFD_RELOC_MIPS_GPREL
Relocation relative to the global pointer.

BFD_RELOC_MIPS_LITERAL
Relocation against a MIPS literal section.

BFD_RELOC_MIPS_GOT16
BFD_RELOC_MIPS_CALL16
BFD_RELOC_MIPS_GPREL32
BFD_RELOC_MIPS_GOT_HI16
BFD_RELOC_MIPS_GOT_LO16
BFD_RELOC_MIPS_CALL_HI16
BFD_RELOC_MIPS_CALL_LO16
MIPS ELF relocations.

BFD_RELOC_386_GOT32
BFD_RELOC_386_PLT32
BFD_RELOC_386_COPY
BFD_RELOC_386_GLOB_DAT
BFD_RELOC_386_JUMP_SLOT
BFD_RELOC_386_RELATIVE
BFD_RELOC_386_GOTOFF
BFD_RELOC_386_GOTPC
i386/elf relocations

BFD_RELOC_NS32K_IMM_8
BFD_RELOC_NS32K_IMM_16
BFD_RELOC_NS32K_IMM_32
BFD_RELOC_NS32K_IMM_8_PCREL
BFD_RELOC_NS32K_IMM_16_PCREL
BFD_RELOC_NS32K_IMM_32_PCREL
BFD_RELOC_NS32K_DISP_8
BFD_RELOC_NS32K_DISP_16
BFD_RELOC_NS32K_DISP_32
BFD_RELOC_NS32K_DISP_8_PCREL
BFD_RELOC_NS32K_DISP_16_PCREL
BFD_RELOC_NS32K_DISP_32_PCREL
ns32k relocations

BFD_RELOC_PPC_B26
BFD_RELOC_PPC_BA26
BFD_RELOC_PPC_TOC16
BFD_RELOC_PPC_B16
BFD_RELOC_PPC_B16_BRTAKEN
BFD_RELOC_PPC_B16_BRNTAKEN
BFD_RELOC_PPC_BA16
BFD_RELOC_PPC_BA16_BRTAKEN
BFD_RELOC_PPC_BA16_BRNTAKEN
BFD_RELOC_PPC_COPY
BFD_RELOC_PPC_GLOB_DAT
BFD_RELOC_PPC_JMP_SLOT
BFD_RELOC_PPC_RELATIVE
BFD_RELOC_PPC_LOCAL24PC
BFD_RELOC_PPC_EMB_NADDR32
BFD_RELOC_PPC_EMB_NADDR16
BFD_RELOC_PPC_EMB_NADDR16_LO
BFD_RELOC_PPC_EMB_NADDR16_HI
BFD_RELOC_PPC_EMB_NADDR16_HA
BFD_RELOC_PPC_EMB_SDAI16
BFD_RELOC_PPC_EMB_SDA2I16
BFD_RELOC_PPC_EMB_SDA2REL
BFD_RELOC_PPC_EMB_SDA21
BFD_RELOC_PPC_EMB_MRKREF
BFD_RELOC_PPC_EMB_RELSEC16
BFD_RELOC_PPC_EMB_RELST_LO
BFD_RELOC_PPC_EMB_RELST_HI
BFD_RELOC_PPC_EMB_RELST_HA
BFD_RELOC_PPC_EMB_BIT_FLD
BFD_RELOC_PPC_EMB_RELSDA
Power(rs6000) and PowerPC relocations.

BFD_RELOC_CTOR

The type of reloc used to build a contructor table - at the moment probably a 32 bit wide absolute relocation, but the target can choose. It generally does map to one of the other relocation types.

BFD_RELOC_ARM_PCREL_BRANCH

ARM 26 bit pc-relative branch. The lowest two bits must be zero and are not stored in the instruction.

BFD_RELOC_ARM_IMMEDIATE
BFD_RELOC_ARM_OFFSET_IMM
BFD_RELOC_ARM_SHIFT_IMM
BFD_RELOC_ARM_SWI
BFD_RELOC_ARM_MULTI
BFD_RELOC_ARM_CP_OFF_IMM
BFD_RELOC_ARM_ADR_IMM
BFD_RELOC_ARM_LDR_IMM
BFD_RELOC_ARM_LITERAL
BFD_RELOC_ARM_IN_POOL

These relocs are only used within the ARM assembler. They are not (at present) written to any object files.

.

```
typedef enum bfd_reloc_code_real bfd_reloc_code_real_type;
```

2.11.0.2 `bfd_reloc_type_lookup`

Synopsis

```
reloc_howto_type *
bfd_reloc_type_lookup (bfd *abfd, bfd_reloc_code_real_type code);
```

Description

Return a pointer to a howto structure which, when invoked, will perform the relocation *code* on data from the architecture noted.

2.11.0.3 `bfd_default_reloc_type_lookup`

Synopsis

```
reloc_howto_type *bfd_default_reloc_type_lookup
   (bfd *abfd, bfd_reloc_code_real_type  code);
```

Description

Provides a default relocation lookup routine for any architecture.

2.11.0.4 `bfd_get_reloc_code_name`

Synopsis

```
const char *bfd_get_reloc_code_name (bfd_reloc_code_real_type code);
```

Description
Provides a printable name for the supplied relocation code. Useful mainly for printing error messages.

2.11.0.5 bfd_generic_relax_section

Synopsis

```
boolean bfd_generic_relax_section
   (bfd *abfd,
    asection *section,
    struct bfd_link_info *,
    boolean *);
```

Description
Provides default handling for relaxing for back ends which don't do relaxing – i.e., does nothing.

2.11.0.6 bfd_generic_get_relocated_section_contents

Synopsis

```
bfd_byte *
bfd_generic_get_relocated_section_contents (bfd *abfd,
    struct bfd_link_info *link_info,
    struct bfd_link_order *link_order,
    bfd_byte *data,
    boolean relocateable,
    asymbol **symbols);
```

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

2.12 Core files

Description
These are functions pertaining to core files.

2.12.0.1 bfd_core_file_failing_command

Synopsis

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

Description
Return a read-only string explaining which program was running when it failed and produced the core file *abfd*.

2.12.0.2 `bfd_core_file_failing_signal`

Synopsis

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

Description

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

2.12.0.3 `core_file_matches_executable_p`

Synopsis

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

Description

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

2.13 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:

- Create a BFD by calling the internal routine `_bfd_new_bfd`, then call `bfd_find_target` with the target string supplied to `bfd_openr` and the new BFD pointer.
- If a null target string was provided to `bfd_find_target`, look up the environment variable `GNUTARGET` and use that as the target string.
- If the target string is still `NULL`, or the target string is `default`, then use the first item in the target vector as the target type, and set `target_defaulted` in the BFD to cause `bfd_check_format` to loop through all the targets. See Section 2.13.1 [bfd_target], page 52. See Section 2.9 [Formats], page 34.
- Otherwise, inspect the elements in the target vector one by one, until a match on target name is found. When found, use it.
- Otherwise return the error `bfd_error_invalid_target` 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. If `target_defaulted` has been set, each possible target type is tried to see if it recognizes the specified format. `bfd_check_format` returns `true` when the caller guesses right.

2.13.1 bfd_target

Description

This structure contains everything that BFD knows about a target. It includes things like its byte order, name, and which routines to call to do various operations. Every BFD points to a target structure with its `xvec` member. The macros below 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. They make the documentation (more) unpleasant to read, so if someone wants to fix this and not break the above, please do.

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

#ifdef DEBUG_BFD_SEND
#undef BFD_SEND
#define BFD_SEND(bfd, message, arglist) \
  (((bfd) && (bfd)->xvec && (bfd)->xvec->message) ? \
    ((*((bfd)->xvec->message)) arglist) : \
    (bfd_assert (__FILE__,__LINE__), NULL))
#endif
```

For operations which index on the BFD format:

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

#ifdef DEBUG_BFD_SEND
#undef BFD_SEND_FMT
#define BFD_SEND_FMT(bfd, message, arglist) \
  (((bfd) && (bfd)->xvec && (bfd)->xvec->message) ? \
   (((bfd)->xvec->message[(int)((bfd)->format)]) arglist) : \
   (bfd_assert (__FILE__,__LINE__), NULL))
#endif
```

This is the structure 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!

```
enum bfd_flavour {
  bfd_target_unknown_flavour,
  bfd_target_aout_flavour,
  bfd_target_coff_flavour,
  bfd_target_ecoff_flavour,
  bfd_target_elf_flavour,
  bfd_target_ieee_flavour,
  bfd_target_nlm_flavour,
  bfd_target_oasys_flavour,
  bfd_target_tekhex_flavour,
  bfd_target_srec_flavour,
```

```
    bfd_target_ihex_flavour,
    bfd_target_som_flavour,
    bfd_target_os9k_flavour,
    bfd_target_amiga_flavour,
    bfd_target_versados_flavour,
    bfd_target_msdos_flavour
  };

  enum bfd_endian { BFD_ENDIAN_BIG, BFD_ENDIAN_LITTLE, BFD_ENDIAN_UNKNOWN };

   /* Forward declaration.  */
  typedef struct bfd_link_info _bfd_link_info;

  typedef struct bfd_target
  {
```

Identifies the kind of target, e.g., SunOS4, Ultrix, etc.

```
    char *name;
```

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

```
    enum bfd_flavour flavour;
```

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

```
    enum bfd_endian byteorder;
```

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

```
    enum bfd_endian header_byteorder;
```

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

```
    flagword object_flags;
```

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 character normally found at the front of a symbol (if any), perhaps '_'.

```
    char symbol_leading_char;
```

The pad character for file names within an archive header.

```
    char ar_pad_char;
```

The maximum number of characters in an archive header.

```
    unsigned short ar_max_namelen;
```

Entries for byte swapping for data. These are different from the other entry points, since they don't take a BFD asthe first argument. Certain other handlers could do the same.

```
    bfd_vma      (*bfd_getx64) PARAMS ((const bfd_byte *));
    bfd_signed_vma (*bfd_getx_signed_64) PARAMS ((const bfd_byte *));
    void         (*bfd_putx64) PARAMS ((bfd_vma, bfd_byte *));
    bfd_vma      (*bfd_getx32) PARAMS ((const bfd_byte *));
    bfd_signed_vma (*bfd_getx_signed_32) PARAMS ((const bfd_byte *));
    void         (*bfd_putx32) PARAMS ((bfd_vma, bfd_byte *));
    bfd_vma      (*bfd_getx16) PARAMS ((const bfd_byte *));
    bfd_signed_vma (*bfd_getx_signed_16) PARAMS ((const bfd_byte *));
    void         (*bfd_putx16) PARAMS ((bfd_vma, bfd_byte *));
```

Byte swapping for the headers

```
    bfd_vma      (*bfd_h_getx64) PARAMS ((const bfd_byte *));
```

```
  bfd_signed_vma (*bfd_h_getx_signed_64) PARAMS ((const bfd_byte *));
  void          (*bfd_h_putx64) PARAMS ((bfd_vma, bfd_byte *));
  bfd_vma       (*bfd_h_getx32) PARAMS ((const bfd_byte *));
  bfd_signed_vma (*bfd_h_getx_signed_32) PARAMS ((const bfd_byte *));
  void          (*bfd_h_putx32) PARAMS ((bfd_vma, bfd_byte *));
  bfd_vma       (*bfd_h_getx16) PARAMS ((const bfd_byte *));
  bfd_signed_vma (*bfd_h_getx_signed_16) PARAMS ((const bfd_byte *));
  void          (*bfd_h_putx16) PARAMS ((bfd_vma, bfd_byte *));
```

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

```
  const struct bfd_target *(*_bfd_check_format[bfd_type_end]) PARAMS ((bfd *));
```

Set the format of a file being written.

```
  boolean             (*_bfd_set_format[bfd_type_end]) PARAMS ((bfd *));
```

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

```
  boolean             (*_bfd_write_contents[bfd_type_end]) PARAMS ((bfd *));
```

The general target vector.

```
   /* Generic entry points.  */
#define BFD_JUMP_TABLE_GENERIC(NAME)\
CAT(NAME,_close_and_cleanup),\
CAT(NAME,_bfd_free_cached_info),\
CAT(NAME,_new_section_hook),\
CAT(NAME,_get_section_contents),\
CAT(NAME,_get_section_contents_in_window)

   /* Called when the BFD is being closed to do any necessary cleanup.  */
  boolean       (*_close_and_cleanup) PARAMS ((bfd *));
   /* Ask the BFD to free all cached information.  */
  boolean (*_bfd_free_cached_info) PARAMS ((bfd *));
   /* Called when a new section is created.  */
  boolean       (*_new_section_hook) PARAMS ((bfd *, sec_ptr));
   /* Read the contents of a section.  */
  boolean       (*_bfd_get_section_contents) PARAMS ((bfd *, sec_ptr, PTR,
                                            file_ptr, bfd_size_type));
  boolean       (*_bfd_get_section_contents_in_window)
                          PARAMS ((bfd *, sec_ptr, bfd_window *,
                                   file_ptr, bfd_size_type));

   /* Entry points to copy private data.  */
#define BFD_JUMP_TABLE_COPY(NAME)\
CAT(NAME,_bfd_copy_private_bfd_data),\
CAT(NAME,_bfd_merge_private_bfd_data),\
CAT(NAME,_bfd_copy_private_section_data),\
CAT(NAME,_bfd_copy_private_symbol_data),\
CAT(NAME,_bfd_set_private_flags),\
CAT(NAME,_bfd_print_private_bfd_data)\
   /* Called to copy BFD general private data from one object file
```

```
     to another.  */
  boolean  (*_bfd_copy_private_bfd_data) PARAMS ((bfd *, bfd *));
   /* Called to merge BFD general private data from one object file
     to a common output file when linking.  */
  boolean  (*_bfd_merge_private_bfd_data) PARAMS ((bfd *, bfd *));
   /* Called to copy BFD private section data from one object file
     to another.  */
  boolean       (*_bfd_copy_private_section_data) PARAMS ((bfd *, sec_ptr,
                                                   bfd *, sec_ptr));
   /* Called to copy BFD private symbol data from one symbol
     to another.  */
  boolean       (*_bfd_copy_private_symbol_data) PARAMS ((bfd *, asymbol *,
   bfd *, asymbol *));
   /* Called to set private backend flags */
  boolean  (*_bfd_set_private_flags) PARAMS ((bfd *, flagword));

   /* Called to print private BFD data */
  boolean       (*_bfd_print_private_bfd_data) PARAMS ((bfd *, PTR));

   /* Core file entry points.  */
#define BFD_JUMP_TABLE_CORE(NAME)\
CAT(NAME,_core_file_failing_command),\
CAT(NAME,_core_file_failing_signal),\
CAT(NAME,_core_file_matches_executable_p)
  char *   (*_core_file_failing_command) PARAMS ((bfd *));
  int      (*_core_file_failing_signal) PARAMS ((bfd *));
  boolean  (*_core_file_matches_executable_p) PARAMS ((bfd *, bfd *));

   /* Archive entry points.  */
#define BFD_JUMP_TABLE_ARCHIVE(NAME)\
CAT(NAME,_slurp_armap),\
CAT(NAME,_slurp_extended_name_table),\
CAT(NAME,_construct_extended_name_table),\
CAT(NAME,_truncate_arname),\
CAT(NAME,_write_armap),\
CAT(NAME,_read_ar_hdr),\
CAT(NAME,_openr_next_archived_file),\
CAT(NAME,_get_elt_at_index),\
CAT(NAME,_generic_stat_arch_elt),\
CAT(NAME,_update_armap_timestamp)
  boolean  (*_bfd_slurp_armap) PARAMS ((bfd *));
  boolean  (*_bfd_slurp_extended_name_table) PARAMS ((bfd *));
  boolean  (*_bfd_construct_extended_name_table)
             PARAMS ((bfd *, char **, bfd_size_type *, const char **));
  void     (*_bfd_truncate_arname) PARAMS ((bfd *, CONST char *, char *));
  boolean  (*write_armap) PARAMS ((bfd *arch,
                              unsigned int elength,
                              struct orl *map,
```

```
                              unsigned int orl_count,
                              int stridx));
  PTR (*_bfd_read_ar_hdr_fn) PARAMS ((bfd *));
  bfd *    (*openr_next_archived_file) PARAMS ((bfd *arch, bfd *prev));
#define bfd_get_elt_at_index(b,i) BFD_SEND(b, _bfd_get_elt_at_index, (b,i))
  bfd *    (*_bfd_get_elt_at_index) PARAMS ((bfd *, symindex));
  int      (*_bfd_stat_arch_elt) PARAMS ((bfd *, struct stat *));
  boolean  (*_bfd_update_armap_timestamp) PARAMS ((bfd *));

   /* Entry points used for symbols.  */
#define BFD_JUMP_TABLE_SYMBOLS(NAME)\
CAT(NAME,_get_symtab_upper_bound),\
CAT(NAME,_get_symtab),\
CAT(NAME,_make_empty_symbol),\
CAT(NAME,_print_symbol),\
CAT(NAME,_get_symbol_info),\
CAT(NAME,_bfd_is_local_label),\
CAT(NAME,_get_lineno),\
CAT(NAME,_find_nearest_line),\
CAT(NAME,_bfd_make_debug_symbol),\
CAT(NAME,_read_minisymbols),\
CAT(NAME,_minisymbol_to_symbol)
  long  (*_bfd_get_symtab_upper_bound) PARAMS ((bfd *));
  long  (*_bfd_canonicalize_symtab) PARAMS ((bfd *,
                                             struct symbol_cache_entry **));
  struct symbol_cache_entry  *
              (*_bfd_make_empty_symbol) PARAMS ((bfd *));
  void          (*_bfd_print_symbol) PARAMS ((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))
  void          (*_bfd_get_symbol_info) PARAMS ((bfd *,
                                      struct symbol_cache_entry *,
                                      symbol_info *));
#define bfd_get_symbol_info(b,p,e) BFD_SEND(b, _bfd_get_symbol_info, (b,p,e))
  boolean  (*_bfd_is_local_label) PARAMS ((bfd *, asymbol *));

  alent *    (*_get_lineno) PARAMS ((bfd *, struct symbol_cache_entry *));
  boolean    (*_bfd_find_nearest_line) PARAMS ((bfd *abfd,
                    struct sec *section, struct symbol_cache_entry **symbols,
                    bfd_vma offset, CONST char **file, CONST char **func,
                    unsigned int *line));
  /* Back-door to allow format-aware applications to create debug symbols
    while using BFD for everything else.  Currently used by the assembler
    when creating COFF files.  */
  asymbol *  (*_bfd_make_debug_symbol) PARAMS ((
       bfd *abfd,
       void *ptr,
```

```
       unsigned long size));
#define bfd_read_minisymbols(b, d, m, s) \
  BFD_SEND (b, _read_minisymbols, (b, d, m, s))
  long  (*_read_minisymbols) PARAMS ((bfd *, boolean, PTR *,
                                      unsigned int *));
#define bfd_minisymbol_to_symbol(b, d, m, f) \
  BFD_SEND (b, _minisymbol_to_symbol, (b, d, m, f))
  asymbol *(*_minisymbol_to_symbol) PARAMS ((bfd *, boolean, const PTR,
                                             asymbol *));

   /* Routines for relocs.  */
#define BFD_JUMP_TABLE_RELOCS(NAME)\
CAT(NAME,_get_reloc_upper_bound),\
CAT(NAME,_canonicalize_reloc),\
CAT(NAME,_bfd_reloc_type_lookup)
  long  (*_get_reloc_upper_bound) PARAMS ((bfd *, sec_ptr));
  long  (*_bfd_canonicalize_reloc) PARAMS ((bfd *, sec_ptr, arelent **,
                                            struct symbol_cache_entry **));
   /* See documentation on reloc types.  */
  reloc_howto_type *
       (*reloc_type_lookup) PARAMS ((bfd *abfd,
                                     bfd_reloc_code_real_type code));

   /* Routines used when writing an object file.  */
#define BFD_JUMP_TABLE_WRITE(NAME)\
CAT(NAME,_set_arch_mach),\
CAT(NAME,_set_section_contents)
  boolean    (*_bfd_set_arch_mach) PARAMS ((bfd *, enum bfd_architecture,
                    unsigned long));
  boolean       (*_bfd_set_section_contents) PARAMS ((bfd *, sec_ptr, PTR,
                                            file_ptr, bfd_size_type));

   /* Routines used by the linker.  */
#define BFD_JUMP_TABLE_LINK(NAME)\
CAT(NAME,_sizeof_headers),\
CAT(NAME,_bfd_get_relocated_section_contents),\
CAT(NAME,_bfd_relax_section),\
CAT(NAME,_bfd_link_hash_table_create),\
CAT(NAME,_bfd_link_add_symbols),\
CAT(NAME,_bfd_final_link),\
CAT(NAME,_bfd_link_split_section)
  int        (*_bfd_sizeof_headers) PARAMS ((bfd *, boolean));
  bfd_byte * (*_bfd_get_relocated_section_contents) PARAMS ((bfd *,
                    struct bfd_link_info *, struct bfd_link_order *,
                    bfd_byte *data, boolean relocateable,
                    struct symbol_cache_entry **));

  boolean    (*_bfd_relax_section) PARAMS ((bfd *, struct sec *,
```

```
                        struct bfd_link_info *, boolean *again));

        /* Create a hash table for the linker.  Different backends store
           different information in this table.  */
       struct bfd_link_hash_table *(*_bfd_link_hash_table_create) PARAMS ((bfd *));

        /* Add symbols from this object file into the hash table.  */
       boolean (*_bfd_link_add_symbols) PARAMS ((bfd *, struct bfd_link_info *));

        /* Do a link based on the link_order structures attached to each
           section of the BFD.  */
       boolean (*_bfd_final_link) PARAMS ((bfd *, struct bfd_link_info *));

        /* Should this section be split up into smaller pieces during linking.  */
       boolean (*_bfd_link_split_section) PARAMS ((bfd *, struct sec *));

       /* Routines to handle dynamic symbols and relocs.  */
     #define BFD_JUMP_TABLE_DYNAMIC(NAME)\
     CAT(NAME,_get_dynamic_symtab_upper_bound),\
     CAT(NAME,_canonicalize_dynamic_symtab),\
     CAT(NAME,_get_dynamic_reloc_upper_bound),\
     CAT(NAME,_canonicalize_dynamic_reloc)
        /* Get the amount of memory required to hold the dynamic symbols. */
       long  (*_bfd_get_dynamic_symtab_upper_bound) PARAMS ((bfd *));
        /* Read in the dynamic symbols.  */
       long  (*_bfd_canonicalize_dynamic_symtab)
         PARAMS ((bfd *, struct symbol_cache_entry **));
        /* Get the amount of memory required to hold the dynamic relocs.  */
       long  (*_bfd_get_dynamic_reloc_upper_bound) PARAMS ((bfd *));
        /* Read in the dynamic relocs.  */
       long  (*_bfd_canonicalize_dynamic_reloc)
         PARAMS ((bfd *, arelent **, struct symbol_cache_entry **));
```

Data for use by back-end routines, which isn't generic enough to belong in this structure.

```
      PTR backend_data;
     } bfd_target;
```

2.13.1.1 bfd_find_target

Synopsis

```
     const bfd_target *bfd_find_target(CONST char *target_name, bfd *abfd);
```

Description

Return a pointer to the transfer vector for the object target named *target_name*. If *target_name* is `NULL`, choose the one in the environment variable `GNUTARGET`; if that is null or not defined, then choose the first entry in the target list. 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.

2.13.1.2 `bfd_target_list`

Synopsis

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

Description

Return a freshly malloced NULL-terminated vector of the names of all the valid BFD targets. Do not modify the names.

2.14 Architectures

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 architecture information is provided by each architecture package. The set of default architectures is selected by the macro `SELECT_ARCHITECTURES`. This is normally set up in the '`config/`*target*`.mt`' 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 architectures 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). BFD's idea of an architecture is implemented in '`archures.c`'.

2.14.1 bfd_architecture

Description

This enum gives the object file's CPU architecture, in a global sense—i.e., what processor family does it belong to? Another field indicates which processor within the family is in use. The machine gives a number which distinguishes 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
#define bfd_mach_i960_jx  7
#define bfd_mach_i960_hx        8

  bfd_arch_a29k,      /* AMD 29000 */
  bfd_arch_sparc,     /* SPARC */
#define bfd_mach_sparc 1
 /* The difference between v8plus and v9 is that v9 is a true 64 bit env.  */
#define bfd_mach_sparc_v8plus 2
#define bfd_mach_sparc_v8plusa 3  /* with ultrasparc add'ns */
#define bfd_mach_sparc_v9 4
#define bfd_mach_sparc_v9a 5  /* with ultrasparc add'ns */
 /* Nonzero if MACH has the v9 instruction set.  */
#define bfd_mach_sparc_v9_p(mach) ((mach) != bfd_mach_sparc)
  bfd_arch_mips,      /* MIPS Rxxxx */
  bfd_arch_i386,      /* Intel 386 */
  bfd_arch_we32k,     /* AT&T WE32xxx */
  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 */
#define bfd_mach_h8300   1
#define bfd_mach_h8300h  2
  bfd_arch_powerpc,   /* PowerPC */
  bfd_arch_rs6000,    /* IBM RS/6000 */
  bfd_arch_hppa,      /* HP PA RISC */
  bfd_arch_z8k,       /* Zilog Z8000 */
#define bfd_mach_z8001 1
#define bfd_mach_z8002 2
  bfd_arch_h8500,     /* Hitachi H8/500 */
  bfd_arch_sh,        /* Hitachi SH */
  bfd_arch_alpha,     /* Dec Alpha */
  bfd_arch_arm,       /* Advanced Risc Machines ARM */
  bfd_arch_ns32k,     /* National Semiconductors ns32000 */
  bfd_arch_w65,       /* WDC 65816 */
```

```
  bfd_arch_last
  };
```

2.14.2 bfd_arch_info

Description
This structure contains information on architectures for use within BFD.

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

  boolean (*scan) PARAMS ((const struct bfd_arch_info *, const char *));

  const struct bfd_arch_info *next;
} bfd_arch_info_type;
```

2.14.2.1 `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 architecture info structure.

2.14.2.2 `bfd_scan_arch`

Synopsis
```
const bfd_arch_info_type *bfd_scan_arch(const char *string);
```

Description
Figure out if BFD supports any cpu which could be described with the name *string*. Return a pointer to an `arch_info` structure if a machine is found, otherwise NULL.

2.14.2.3 `bfd_arch_get_compatible`

Synopsis
```
const bfd_arch_info_type *bfd_arch_get_compatible(
    const bfd *abfd,
    const bfd *bbfd);
```
Description
Determine whether two BFDs' architectures and machine types are compatible. 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.

2.14.2.4 `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 const bfd_arch_info_type bfd_default_arch_struct;
```

2.14.2.5 `bfd_set_arch_info`

Synopsis
```
void bfd_set_arch_info(bfd *abfd, const bfd_arch_info_type *arg);
```
Description
Set the architecture info of *abfd* to *arg*.

2.14.2.6 `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 BFD *abfd* to *arch* and *mach*. Find the correct pointer to a structure and insert it into the `arch_info` pointer.

2.14.2.7 `bfd_get_arch`

Synopsis

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

Description

Return the enumerated type which describes the BFD *abfd*'s architecture.

2.14.2.8 `bfd_get_mach`

Synopsis

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

Description

Return the long type which describes the BFD *abfd*'s machine.

2.14.2.9 `bfd_arch_bits_per_byte`

Synopsis

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

Description

Return the number of bits in one of the BFD *abfd*'s architecture's bytes.

2.14.2.10 `bfd_arch_bits_per_address`

Synopsis

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

Description

Return the number of bits in one of the BFD *abfd*'s architecture's addresses.

2.14.2.11 `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.

2.14.2.12 `bfd_default_scan`

Synopsis

```
boolean bfd_default_scan(const struct bfd_arch_info *info, const char *string);
```

Description

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

2.14.2.13 bfd_get_arch_info

Synopsis

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

Description

Return the architecture info struct in *abfd*.

2.14.2.14 bfd_lookup_arch

Synopsis

```
const bfd_arch_info_type *bfd_lookup_arch
   (enum bfd_architecture
    arch,
    unsigned long machine);
```

Description

Look for the architecure info structure which matches the arguments *arch* and *machine*. A machine of 0 matches the machine/architecture structure which marks itself as the default.

2.14.2.15 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. This routine is depreciated.

2.15 Opening and closing BFDs

2.15.0.1 bfd_openr

Synopsis

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

Description

Open the file *filename* (using `fopen`) with the target *target*. Return a pointer to the created BFD. Calls `bfd_find_target`, so *target* is interpreted as by that function. If `NULL` is returned then an error has occured. Possible errors are `bfd_error_no_memory`, `bfd_error_invalid_target` or `system_call` error.

2.15.0.2 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. When the file is later `bfd_close`d, the file descriptor will be closed. If the caller desires that this file descriptor be cached by BFD (opened as needed, closed as needed to free descriptors for other opens), with the supplied *fd* used as an initial file descriptor (but subject to closure at any time), call bfd_set_cacheable(bfd, 1) on the returned BFD. The default is to assume no cacheing; the file descriptor will remain open until `bfd_close`, and will not be affected by BFD operations on other files. Possible errors are `bfd_error_no_memory`, `bfd_error_invalid_target` and `bfd_error_system_call`.

2.15.0.3 bfd_openstreamr

Synopsis

```
bfd *bfd_openstreamr();
```

Description

Open a BFD for read access on an existing stdio stream. When the BFD is passed to `bfd_close`, the stream will be closed.

2.15.0.4 bfd_openw

Synopsis

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

Description

Create a BFD, associated with file *filename*, using the file format *target*, and return a pointer to it. Possible errors are `bfd_error_system_call`, `bfd_error_no_memory`, `bfd_error_invalid_target`.

2.15.0.5 bfd_close

Synopsis

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

Description

Close 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. The file descriptor associated with the BFD is closed (even if it was passed in to BFD by `bfd_fdopenr`).

Returns

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

2.15.0.6 `bfd_close_all_done`

Synopsis

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

Description

Close a BFD. 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 *abfd*.

2.15.0.7 `bfd_create`

Synopsis

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

Description

Create 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`.

2.15.0.8 `bfd_alloc_by_size_t`

Synopsis

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

Description

Allocate a block of *wanted* bytes of memory in the obstack attatched to `abfd` and return a pointer to it.

2.16 Internal functions

Description

These routines are used within BFD. They are not intended for export, but are documented here for completeness.

2.16.0.1 `bfd_write_bigendian_4byte_int`

Synopsis

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

Description

Write a 4 byte integer *i* to the output BFD *abfd*, in big endian order regardless of what else is going on. This is useful in archives.

2.16.0.2 `bfd_put_size`

2.16.0.3 `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. Note that types accepted and returned by these macros are identical so they can be swapped around in macros—for example, '`libaout.h`' defines `GET_WORD` to either `bfd_get_32` or `bfd_get_64`. In the put routines, *val* must be a `bfd_vma`. If we are on a system without prototypes, the caller is responsible for making sure that is true, with a cast if necessary. We don't cast them in the macro definitions because that would prevent `lint` or `gcc -Wall` from detecting sins such as passing a pointer. To detect calling these with less than a `bfd_vma`, use `gcc -Wconversion` on a host with 64 bit `bfd_vma`'s.

```
 /* Byte swapping macros for user section data.  */

#define bfd_put_8(abfd, val, ptr) \
                (*((unsigned char *)(ptr)) = (unsigned char)(val))
#define bfd_put_signed_8 \
bfd_put_8
#define bfd_get_8(abfd, ptr) \
                (*(unsigned char *)(ptr))
#define bfd_get_signed_8(abfd, ptr) \
((*(unsigned char *)(ptr) ^ 0x80) - 0x80)

#define bfd_put_16(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx16, ((val),(ptr)))
#define bfd_put_signed_16 \
 bfd_put_16
#define bfd_get_16(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx16, (ptr))
#define bfd_get_signed_16(abfd, ptr) \
           BFD_SEND (abfd, bfd_getx_signed_16, (ptr))

#define bfd_put_32(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx32, ((val),(ptr)))
#define bfd_put_signed_32 \
```

```
 bfd_put_32
#define bfd_get_32(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx32, (ptr))
#define bfd_get_signed_32(abfd, ptr) \
 BFD_SEND(abfd, bfd_getx_signed_32, (ptr))

#define bfd_put_64(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx64, ((val), (ptr)))
#define bfd_put_signed_64 \
 bfd_put_64
#define bfd_get_64(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx64, (ptr))
#define bfd_get_signed_64(abfd, ptr) \
 BFD_SEND(abfd, bfd_getx_signed_64, (ptr))
```

2.16.0.4 bfd_h_put_size

Description
These macros have the same function as their `bfd_get_x` bretheren, 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 endian order.

```
 /* Byte swapping macros for file header data.  */

#define bfd_h_put_8(abfd, val, ptr) \
bfd_put_8 (abfd, val, ptr)
#define bfd_h_put_signed_8(abfd, val, ptr) \
bfd_put_8 (abfd, val, ptr)
#define bfd_h_get_8(abfd, ptr) \
bfd_get_8 (abfd, ptr)
#define bfd_h_get_signed_8(abfd, ptr) \
bfd_get_signed_8 (abfd, ptr)

#define bfd_h_put_16(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_h_putx16,(val,ptr))
#define bfd_h_put_signed_16 \
 bfd_h_put_16
#define bfd_h_get_16(abfd, ptr) \
                BFD_SEND(abfd, bfd_h_getx16,(ptr))
#define bfd_h_get_signed_16(abfd, ptr) \
 BFD_SEND(abfd, bfd_h_getx_signed_16, (ptr))

#define bfd_h_put_32(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_h_putx32,(val,ptr))
#define bfd_h_put_signed_32 \
```

```
 bfd_h_put_32
#define bfd_h_get_32(abfd, ptr) \
                BFD_SEND(abfd, bfd_h_getx32,(ptr))
#define bfd_h_get_signed_32(abfd, ptr) \
 BFD_SEND(abfd, bfd_h_getx_signed_32, (ptr))

#define bfd_h_put_64(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_h_putx64,(val, ptr))
#define bfd_h_put_signed_64 \
 bfd_h_put_64
#define bfd_h_get_64(abfd, ptr) \
                BFD_SEND(abfd, bfd_h_getx64,(ptr))
#define bfd_h_get_signed_64(abfd, ptr) \
 BFD_SEND(abfd, bfd_h_getx_signed_64, (ptr))
```

2.16.0.5 `bfd_log2`

Synopsis

```
unsigned int bfd_log2(bfd_vma x);
```

Description

Return the log base 2 of the value supplied, rounded up. E.g., an *x* of 1025 returns 11.

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 maximum 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

Check to see if the required BFD is the same as the last one looked up. If so, then it can use the stream 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

```
boolean bfd_cache_init (bfd *abfd);
```

Description

Add a newly opened BFD to the cache.

2.17.0.5 `bfd_cache_close`

Synopsis

```
boolean bfd_cache_close (bfd *abfd);
```

Description

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

Returns

`false` is returned if closing the file fails, `true` is returned if all is well.

2.17.0.6 `bfd_open_file`

Synopsis

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

Description

Call the OS to open a file for *abfd*. Return the `FILE *` (possibly `NULL`) that results from this operation. Set up the BFD so that future accesses know the file is open. If the `FILE *` returned is `NULL`, then it won't have been put in the cache, so it won't have to be removed from it.

2.17.0.7 `bfd_cache_lookup_worker`

Synopsis

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

Description

Called when the macro `bfd_cache_lookup` fails to find a quick answer. Find a file descriptor for *abfd*. If necessary, it open it. If there are already more than `BFD_CACHE_MAX_OPEN` files open, it tries to close one first, to avoid running out of file descriptors.

2.18 Linker Functions

The linker uses three special entry points in the BFD target vector. It is not necessary to write special routines for these entry points when creating a new BFD back end, since generic versions are provided. However, writing them can speed up linking and make it use significantly less runtime memory. The first routine creates a hash table used by the other routines. The second routine adds the symbols from an object file to the hash table. The third routine takes all the object files and links them together to create the output file. These routines are designed so that the linker proper does not need to know anything about the symbols in the object files that it is linking. The linker merely arranges the sections as directed by the linker script and lets BFD handle the details of symbols and relocs. The second routine and third routines are passed a pointer to a `struct bfd_link_info` structure (defined in `bfdlink.h`) which holds information relevant to the link, including the linker hash table (which was created by the first routine) and a set of callback functions to the linker proper. The generic linker routines are in `linker.c`, and use the header file `genlink.h`. As of this writing, the only back ends which have implemented versions of these routines are a.out (in `aoutx.h`) and ECOFF (in `ecoff.c`). The a.out routines are used as examples throughout this section.

2.18.1 Creating a linker hash table

The linker routines must create a hash table, which must be derived from `struct bfd_link_hash_table` described in `bfdlink.c`. See Section 2.19 [Hash Tables], page 75 for information on how to create a derived hash table. This entry point is called using the target vector of the linker output file. The `_bfd_link_hash_table_create` entry point must allocate and initialize an instance of the desired hash table. If the back end does not require any additional information to be stored with the entries in the hash table, the entry point may simply create a `struct bfd_link_hash_table`. Most likely, however, some additional information will be needed. For example, with each entry in the hash table the a.out linker keeps the index the symbol has in the final output file (this index number is used so that when doing a relocateable link the symbol index used in the output file can be quickly filled in when copying over a reloc). The a.out linker code defines the required structures and functions for a hash table derived from `struct bfd_link_hash_table`. The a.out linker hash table is created by the function `NAME(aout,link_hash_table_create)`; it simply allocates space for the hash table, initializes it, and returns a pointer to it. When writing the linker routines for a new back end, you will generally not know exactly which fields will be required until you have finished. You should simply create a new hash table which defines no additional fields, and then simply add fields as they become necessary.

2.18.2 Adding symbols to the hash table

The linker proper will call the `_bfd_link_add_symbols` entry point for each object file or archive which is to be linked (typically these are the files named on the command line, but some may also come from the linker script). The entry point is responsible for examining the file. For an object file, BFD must add any relevant symbol information to the hash table. For an archive, BFD must determine which elements of the archive should be used and adding them to the link. The a.out version of this entry point is `NAME(aout,link_add_symbols)`.

2.18.2.1 Differing file formats

Normally all the files involved in a link will be of the same format, but it is also possible to link together different format object files, and the back end must support that. The `_bfd_link_add_symbols` entry point is called via the target vector of the file to be added. This has an important consequence: the function may not assume that the hash table is the type created by the corresponding `_bfd_link_hash_table_create` vector. All the `_bfd_link_add_symbols` function can assume about the hash table is that it is derived from `struct bfd_link_hash_table`.

Sometimes the `_bfd_link_add_symbols` function must store some information in the hash table entry to be used by the `_bfd_final_link` function. In such a case the `creator` field of the hash table must be checked to make sure that the hash table was created by an object file of the same format.

The `_bfd_final_link` routine must be prepared to handle a hash entry without any extra information added by the `_bfd_link_add_symbols` function. A hash entry without extra information will also occur when the linker script directs the linker to create a symbol. Note that, regardless of how a hash table entry is added, all the fields will be initialized to some sort of null value by the hash table entry initialization function.

See `ecoff_link_add_externals` for an example of how to check the `creator` field before saving information (in this case, the ECOFF external symbol debugging information) in a hash table entry.

2.18.2.2 Adding symbols from an object file

When the `_bfd_link_add_symbols` routine is passed an object file, it must add all externally visible symbols in that object file to the hash table. The actual work of adding the symbol to the hash table is normally handled by the function `_bfd_generic_link_add_one_symbol`. The `_bfd_link_add_symbols` routine is responsible for reading all the symbols from the object file and passing the correct information to `_bfd_generic_link_add_one_symbol`.

The `_bfd_link_add_symbols` routine should not use `bfd_canonicalize_symtab` to read the symbols. The point of providing this routine is to avoid the overhead of converting the symbols into generic `asymbol` structures.

`_bfd_generic_link_add_one_symbol` handles the details of combining common symbols, warning about multiple definitions, and so forth. It takes arguments which describe the symbol to add, notably symbol flags, a section, and an offset. The symbol flags include such things as `BSF_WEAK` or `BSF_INDIRECT`. The section is a section in the object file, or something like `bfd_und_section_ptr` for an undefined symbol or bfd_com_section_ptr for a common symbol. If the _bfd_final_link routine is also going to need to read the symbol information, the _bfd_link_add_symbols routine should save it somewhere attached to the object file BFD. However, the information should only be saved if the keep_memory field of the info argument is true, so that the -no-keep-memory linker switch is effective. The a.out function which adds symbols from an object file is aout_link_add_object_symbols, and most of the interesting work is in aout_link_add_symbols. The latter saves pointers to the hash tables entries created by _bfd_generic_link_add_one_symbol indexed by symbol number, so that the _bfd_final_link routine does not have to call the hash table lookup routine to locate the entry.

2.18.2.3 Adding symbols from an archive

When the _bfd_link_add_symbols routine is passed an archive, it must look through the symbols defined by the archive and decide which elements of the archive should be included in the link. For each such element it must call the add_archive_element linker callback, and it must add the symbols from the object file to the linker hash table. In most cases the work of looking through the symbols in the archive should be done by the _bfd_generic_link_add_archive_symbols function. This function builds a hash table from the archive symbol table and looks through the list of undefined symbols to see which elements should be included. _bfd_generic_link_add_archive_symbols is passed a function to call to make the final decision about adding an archive element to the link and to do the actual work of adding the symbols to the linker hash table. The function passed to _bfd_generic_link_add_archive_symbols must read the symbols of the archive element and decide whether the archive element should be included in the link. If the element is to be included, the add_archive_element linker callback routine must be called with the element as an argument, and the elements symbols must be added to the linker hash table just as though the element had itself been passed to the _bfd_link_add_symbols function. When the a.out _bfd_link_add_symbols function receives an archive, it calls _bfd_generic_link_add_archive_symbols passing aout_link_check_archive_element as the function argument. aout_link_check_archive_element calls aout_link_check_ar_symbols. If the latter decides to add the element (an element is only added if it provides a real, non-common, definition for a previously undefined or common symbol) it calls the add_archive_element callback and then aout_link_check_archive_element calls aout_link_add_symbols to actually add the symbols to the linker hash table. The ECOFF back end is unusual in that it does not normally call _bfd_generic_link_add_archive_symbols, because ECOFF archives already contain a hash table of symbols. The ECOFF back end searches the archive itself to avoid the overhead of creating a new hash table.

2.18.3 Performing the final link

When all the input files have been processed, the linker calls the _bfd_final_link entry point of the output BFD. This routine is responsible for producing the final output file, which has several aspects. It must relocate the contents of the input sections and copy the data into the output sections. It must build an output symbol table including any local symbols from the input files and the global symbols from the hash table. When producing relocateable output, it must modify the input relocs and write them into the output file. There may also be object format dependent work to be done. The linker will also call the `write_object_contents` entry point when the BFD is closed. The two entry points must work together in order to produce the correct output file. The details of how this works are inevitably dependent upon the specific object file format. The a.out `_bfd_final_link` routine is `NAME(aout,final_link)`.

2.18.3.1 Information provided by the linker

Before the linker calls the `_bfd_final_link` entry point, it sets up some data structures for the function to use. The `input_bfds` field of the `bfd_link_info` structure will point to a list of all the input files included in the link. These files are linked through the `link_next` field of the `bfd` structure. Each section in the output file will have a list of `link_order` structures attached to the `link_order_head` field (the `link_order` structure is defined in `bfdlink.h`). These structures describe how to create the contents of the output section in terms of the contents of various input sections, fill constants, and, eventually, other types of information. They also describe relocs that must be created by the BFD backend, but do not correspond to any input file; this is used to support -Ur, which builds constructors while generating a relocateable object file.

2.18.3.2 Relocating the section contents

The `_bfd_final_link` function should look through the `link_order` structures attached to each section of the output file. Each `link_order` structure should either be handled specially, or it should be passed to the function `_bfd_default_link_order` which will do the right thing (`_bfd_default_link_order` is defined in `linker.c`). For efficiency, a `link_order` of type `bfd_indirect_link_order` whose associated section belongs to a BFD of the same format as the output BFD must be handled specially. This type of `link_order` describes part of an output section in terms of a section belonging to one of the input files. The `_bfd_final_link` function should read the contents of the section and any associated relocs, apply the relocs to the section contents, and write out the modified section contents. If performing a relocateable link, the relocs themselves must also be modified and written out. The functions `_bfd_relocate_contents` and `_bfd_final_link_relocate` provide some general support for performing the actual relocations, notably overflow checking. Their arguments include information about the symbol the relocation is against and a `reloc_howto_type` argument which describes the relocation to perform. These functions are defined in `reloc.c`. The a.out function which handles reading, relocating, and writing section contents is `aout_link_input_section`. The actual relocation is done in `aout_link_input_section_std` and `aout_link_input_section_ext`.

2.18.3.3 Writing the symbol table

The `_bfd_final_link` function must gather all the symbols in the input files and write them out. It must also write out all the symbols in the global hash table. This must be controlled by the `strip` and `discard` fields of the `bfd_link_info` structure. The local symbols of the input files will not have been entered into the linker hash table. The `_bfd_final_link` routine must consider

each input file and include the symbols in the output file. It may be convenient to do this when looking through the `link_order` structures, or it may be done by stepping through the `input_bfds` list. The `_bfd_final_link` routine must also traverse the global hash table to gather all the externally visible symbols. It is possible that most of the externally visible symbols may be written out when considering the symbols of each input file, but it is still necessary to traverse the hash table since the linker script may have defined some symbols that are not in any of the input files. The `strip` field of the `bfd_link_info` structure controls which symbols are written out. The possible values are listed in `bfdlink.h`. If the value is `strip_some`, then the `keep_hash` field of the `bfd_link_info` structure is a hash table of symbols to keep; each symbol should be looked up in this hash table, and only symbols which are present should be included in the output file. If the `strip` field of the `bfd_link_info` structure permits local symbols to be written out, the `discard` field is used to further controls which local symbols are included in the output file. If the value is `discard_l`, then all local symbols which begin with a certain prefix are discarded; this prefix is described by the `lprefix` and `lprefix_len` fields of the `bfd_link_info` structure.
The a.out backend handles symbols by calling `aout_link_write_symbols` on each input BFD and then traversing the global hash table with the function `aout_link_write_other_symbol`. It builds a string table while writing out the symbols, which is written to the output file at the end of `NAME(aout,final_link)`.

2.18.3.4 `bfd_link_split_section`

Synopsis

```
boolean bfd_link_split_section(bfd *abfd, asection *sec);
```

Description

Return nonzero if *sec* should be split during a reloceatable or final link.

```
#define bfd_link_split_section(abfd, sec) \
       BFD_SEND (abfd, _bfd_link_split_section, (abfd, sec))
```

2.19 Hash Tables

BFD provides a simple set of hash table functions. Routines are provided to initialize a hash table, to free a hash table, to look up a string in a hash table and optionally create an entry for it, and to traverse a hash table. There is currently no routine to delete an string from a hash table. The basic hash table does not permit any data to be stored with a string. However, a hash table is designed to present a base class from which other types of hash tables may be derived. These derived types may store additional information with the string. Hash tables were implemented in this way, rather than simply providing a data pointer in a hash table entry, because they were designed for use by the linker back ends. The linker may create thousands of hash table entries, and the overhead of allocating private data and storing and following pointers becomes noticeable. The basic hash table code is in `hash.c`.

2.19.1 Creating and freeing a hash table

To create a hash table, create an instance of a `struct bfd_hash_table` (defined in `bfd.h`) and call `bfd_hash_table_init` (if you know approximately how many entries you will need, the function `bfd_hash_table_init_n`, which takes a *size* argument, may be used). `bfd_hash_table_init` returns `false` if some sort of error occurs. The function `bfd_hash_table_init` take as an argument a function to use to create new entries. For a basic hash table, use the function `bfd_hash_newfunc`. See Section 2.19.4 [Deriving a New Hash Table Type], page 76 for why you would want to use a different value for this argument. `bfd_hash_table_init` will create an obstack which will be used to allocate new entries. You may allocate memory on this obstack using `bfd_hash_allocate`. Use `bfd_hash_table_free` to free up all the memory that has been allocated for a hash table. This will not free up the `struct bfd_hash_table` itself, which you must provide.

2.19.2 Looking up or entering a string

The function `bfd_hash_lookup` is used both to look up a string in the hash table and to create a new entry. If the *create* argument is `false`, `bfd_hash_lookup` will look up a string. If the string is found, it will returns a pointer to a `struct bfd_hash_entry`. If the string is not found in the table `bfd_hash_lookup` will return `NULL`. You should not modify any of the fields in the returns `struct bfd_hash_entry`. If the *create* argument is `true`, the string will be entered into the hash table if it is not already there. Either way a pointer to a `struct bfd_hash_entry` will be returned, either to the existing structure or to a newly created one. In this case, a `NULL` return means that an error occurred. If the *create* argument is `true`, and a new entry is created, the *copy* argument is used to decide whether to copy the string onto the hash table obstack or not. If *copy* is passed as `false`, you must be careful not to deallocate or modify the string as long as the hash table exists.

2.19.3 Traversing a hash table

The function `bfd_hash_traverse` may be used to traverse a hash table, calling a function on each element. The traversal is done in a random order. `bfd_hash_traverse` takes as arguments a function and a generic `void *` pointer. The function is called with a hash table entry (a `struct bfd_hash_entry *`) and the generic pointer passed to `bfd_hash_traverse`. The function must return a `boolean` value, which indicates whether to continue traversing the hash table. If the function returns `false`, `bfd_hash_traverse` will stop the traversal and return immediately.

2.19.4 Deriving a new hash table type

Many uses of hash tables want to store additional information which each entry in the hash table. Some also find it convenient to store additional information with the hash table itself. This may be done using a derived hash table. Since C is not an object oriented language, creating a derived hash table requires sticking together some boilerplate routines with a few differences specific to the type of hash table you want to create. An example of a derived hash table is the linker hash table. The structures for this are defined in `bfdlink.h`. The functions are in `linker.c`. You may also derive a hash table from an already derived hash table. For example, the a.out linker backend code uses a hash table derived from the linker hash table.

2.19.4.1 Define the derived structures

You must define a structure for an entry in the hash table, and a structure for the hash table itself. The first field in the structure for an entry in the hash table must be of the type used for an entry in the hash table you are deriving from. If you are deriving from a basic hash table this is `struct bfd_hash_entry`, which is defined in `bfd.h`. The first field in the structure for the hash table itself must be of the type of the hash table you are deriving from itself. If you are deriving from a basic hash table, this is `struct bfd_hash_table`. For example, the linker hash table defines `struct bfd_link_hash_entry` (in `bfdlink.h`). The first field, `root`, is of type `struct bfd_hash_entry`. Similarly, the first field in `struct bfd_link_hash_table`, `table`, is of type `struct bfd_hash_table`.

2.19.4.2 Write the derived creation routine

You must write a routine which will create and initialize an entry in the hash table. This routine is passed as the function argument to `bfd_hash_table_init`. In order to permit other hash tables to be derived from the hash table you are creating, this routine must be written in a standard way. The first argument to the creation routine is a pointer to a hash table entry. This may be `NULL`, in which case the routine should allocate the right amount of space. Otherwise the space has already been allocated by a hash table type derived from this one. After allocating space, the creation routine must call the creation routine of the hash table type it is derived from, passing in a pointer to the space it just allocated. This will initialize any fields used by the base hash table. Finally the creation routine must initialize any local fields for the new hash table type. Here is a boilerplate example of a creation routine. *function_name* is the name of the routine. *entry_type* is the type of an entry in the hash table you are creating. *base_newfunc* is the name of the creation routine of the hash table type your hash table is derived from.

.struct bfd_hash_entry *

```
     function_name (entry, table, string)
          struct bfd_hash_entry *entry;
          struct bfd_hash_table *table;
          const char *string;
     {
       struct entry_type *ret = (entry_type *) entry;

      /* Allocate the structure if it has not already been allocated by a
         derived class.  */
       if (ret == (entry_type *) NULL)
         {
           ret = ((entry_type *)
           bfd_hash_allocate (table, sizeof (entry_type)));
           if (ret == (entry_type *) NULL)
             return NULL;
```

```
    }

  /* Call the allocation method of the base class.  */
   ret = ((entry_type *)
  base_newfunc ((struct bfd_hash_entry *) ret, table, string));

  /* Initialize the local fields here.  */

   return (struct bfd_hash_entry *) ret;
}
```

Description

The creation routine for the linker hash table, which is in `linker.c`, looks just like this example. *function_name* is `_bfd_link_hash_newfunc`. *entry_type* is `struct bfd_link_hash_entry`. *base_newfunc* is `bfd_hash_newfunc`, the creation routine for a basic hash table. `_bfd_link_hash_newfunc` also initializes the local fields in a linker hash table entry: `type`, `written` and `next`.

2.19.4.3 Write other derived routines

You will want to write other routines for your new hash table, as well. You will want an initialization routine which calls the initialization routine of the hash table you are deriving from and initializes any other local fields. For the linker hash table, this is `_bfd_link_hash_table_init` in `linker.c`. You will want a lookup routine which calls the lookup routine of the hash table you are deriving from and casts the result. The linker hash table uses `bfd_link_hash_lookup` in `linker.c` (this actually takes an additional argument which it uses to decide how to return the looked up value). You may want a traversal routine. This should just call the traversal routine of the hash table you are deriving from with appropriate casts. The linker hash table uses `bfd_link_hash_traverse` in `linker.c`. These routines may simply be defined as macros. For example, the a.out backend linker hash table, which is derived from the linker hash table, uses macros for the lookup and traversal routines. These are `aout_link_hash_lookup` and `aout_link_hash_traverse` in aoutx.h.

3 BFD back ends

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

3.1.1 Relocations

Description
The file '`aoutx.h`' provides for both the *standard* and *extended* forms of a.out relocation records.
The standard records contain 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`

Synopsis

```
void aout_size_swap_exec_header_in,
   (bfd *abfd,
    struct external_exec *raw_bytes,
    struct internal_exec *execp);
```

Description
Swap the information in an executable header *raw_bytes* taken from a raw byte stream memory image into the internal exec header structure *execp*.

3.1.2.2 `aout_`*size*`_swap_exec_header_out`

Synopsis

```
void aout_size_swap_exec_header_out
   (bfd *abfd,
    struct internal_exec *execp,
    struct external_exec *raw_bytes);
```

Description
Swap the information in an internal exec header structure *execp* into the buffer *raw_bytes* ready

for writing to disk.

3.1.2.3 aout_*size*_some_aout_object_p

Synopsis
```
const bfd_target *aout_size_some_aout_object_p
   (bfd *abfd,
    const bfd_target *(*callback_to_real_object_p)());
```
Description
Some a.out variant thinks that the file open in *abfd* checking is an a.out file. Do some more checking, and set up for access if it really is. Call back to the calling environment's "finish up" function just before returning, to handle any last-minute setup.

3.1.2.4 aout_*size*_mkobject

Synopsis
```
boolean aout_size_mkobject, (bfd *abfd);
```
Description
Initialize BFD *abfd* for use with a.out files.

3.1.2.5 aout_*size*_machine_type

Synopsis
```
enum machine_type  aout_size_machine_type
   (enum bfd_architecture arch,
    unsigned long machine));
```

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

3.1.2.6 aout_*size*_set_arch_mach

Synopsis
```
boolean aout_size_set_arch_mach,
   (bfd *,
    enum bfd_architecture arch,
    unsigned long machine));
```

Description
Set the architecture and the machine of the BFD *abfd* to the values *arch* and *machine*. Verify that

abfd's format can support the architecture required.

3.1.2.7 `aout_size_new_section_hook`

Synopsis

```
boolean aout_size_new_section_hook,
   (bfd *abfd,
    asection *newsect));
```

Description

Called by the BFD in response to a `bfd_make_section` request.

3.2 coff backends

BFD supports a number of different flavours of coff format. The major differences between formats are the sizes and alignments of fields in structures on disk, and the occasional extra field. Coff in all its varieties is implemented 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 `#include`s '`coff/m88k.h`' which defines the external structure of the coff format for the 88k, and '`coff/internal.h`' which defines the internal structure. '`coff-m88k.c`' also defines the relocations used by the 88k format See Section 2.10 [Relocations], page 36. 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 implementations the version of coff which is most like the one you want to use. For example, we'll say that i386 coff is the one you select, and that your coff flavour is called foo. Copy '`i386coff.c`' to '`foocoff.c`', copy '`../include/coff/i386.h`' to '`../include/coff/foo.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/coff/foo.h`' so that they match what you need. You will probably also have to add `#ifdef`s to the code in '`coff/internal.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 and your host system's 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 File layout

The Coff backend is split into generic routines that are applicable to any Coff target and routines that are specific to a particular target. The target-specific routines are further split into ones which are basically the same for all Coff targets except that they use the external symbol format or use different values for certain constants. The generic routines are in 'coffgen.c'. These routines work for any Coff target. They use some hooks into the target specific code; the hooks are in a bfd_coff_backend_data structure, one of which exists for each target. The essentially similar target-specific routines are in 'coffcode.h'. This header file includes executable C code. The various Coff targets first include the appropriate Coff header file, make any special defines that are needed, and then include 'coffcode.h'. Some of the Coff targets then also have additional routines in the target source file itself. For example, 'coff-i960.c' includes 'coff/internal.h' and 'coff/i960.h'. It then defines a few constants, such as I960, and includes 'coffcode.h'. Since the i960 has complex relocation types, 'coff-i960.c' also includes some code to manipulate the i960 relocs. This code is not in 'coffcode.h' because it would not be used by any other target.

3.2.2.2 Bit twiddling

Each flavour of coff supported in BFD has its own header file describing the external layout of the structures. There is also an internal description of the coff layout, in 'coff/internal.h'. 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'. E.g., 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 to 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. Doing so also allows BFD (and thus the linker) to use the same header files as gas, which makes one avenue to disaster disappear.

3.2.2.3 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 problem 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 coff_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 change 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 at 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 Section 2.7 [Symbols], page 26. 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.

3.2.2.4 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 the symbol was taken, 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. This transformation uses the `output_section` field of the `asymbol`'s `asection` See Section 2.6 [Sections], page 16.

- `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 into the index into the symbol table of the asymbol.

- `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 table to the file.

3.2.2.5 coff_symbol_type

Description

The hidden information for an asymbol is described in 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 value of this symbol be renumbered.  Used for
          XCOFF C_BSTAT symbols.  Set by coff_slurp_symbol_table.  */
unsigned int fix_value : 1;

       /* Should the tag field of this symbol be renumbered.
          Created by coff_pointerize_aux. */
unsigned int fix_tag : 1;

       /* Should the endidx field of this symbol be renumbered.
          Created by coff_pointerize_aux. */
unsigned int fix_end : 1;

       /* Should the x_csect.x_scnlen field be renumbered.
          Created by coff_pointerize_aux. */
unsigned int fix_scnlen : 1;

       /* Fix up an XCOFF C_BINCL/C_EINCL symbol.  The value is the
          index into the line number entries.  Set by
          coff_slurp_symbol_table.  */
unsigned int fix_line : 1;

       /* 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;
```

3.2.2.6 `bfd_coff_backend_data`

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

```
typedef struct
{
  void (*_bfd_coff_swap_aux_in) PARAMS ((
       bfd            *abfd,
       PTR             ext,
       int             type,
       int             class,
       int             indaux,
       int             numaux,
       PTR             in));

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

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

Special entry points for gas to swap out coff parts:

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

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

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

       unsigned int (*_bfd_coff_swap_reloc_out) PARAMS ((
             bfd     *abfd,
            PTR src,
      PTR dst));

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

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

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

Special entry points for generic COFF routines to call target dependent COFF routines:

```
       unsigned int _bfd_filhsz;
       unsigned int _bfd_aoutsz;
       unsigned int _bfd_scnhsz;
       unsigned int _bfd_symesz;
       unsigned int _bfd_auxesz;
       unsigned int _bfd_relsz;
       unsigned int _bfd_linesz;
       boolean _bfd_coff_long_filenames;
       void (*_bfd_coff_swap_filehdr_in) PARAMS ((
             bfd     *abfd,
             PTR     ext,
             PTR     in));
       void (*_bfd_coff_swap_aouthdr_in) PARAMS ((
             bfd     *abfd,
             PTR     ext,
             PTR     in));
       void (*_bfd_coff_swap_scnhdr_in) PARAMS ((
             bfd     *abfd,
             PTR     ext,
             PTR     in));
       void (*_bfd_coff_swap_reloc_in) PARAMS ((
```

```
      bfd     *abfd,
      PTR     ext,
      PTR     in));
boolean (*_bfd_coff_bad_format_hook) PARAMS ((
      bfd     *abfd,
      PTR     internal_filehdr));
boolean (*_bfd_coff_set_arch_mach_hook) PARAMS ((
      bfd     *abfd,
      PTR     internal_filehdr));
PTR (*_bfd_coff_mkobject_hook) PARAMS ((
      bfd     *abfd,
      PTR     internal_filehdr,
      PTR     internal_aouthdr));
flagword (*_bfd_styp_to_sec_flags_hook) PARAMS ((
      bfd     *abfd,
      PTR     internal_scnhdr,
      const char *name));
void (*_bfd_set_alignment_hook) PARAMS ((
      bfd     *abfd,
      asection *sec,
      PTR     internal_scnhdr));
boolean (*_bfd_coff_slurp_symbol_table) PARAMS ((
      bfd     *abfd));
boolean (*_bfd_coff_symname_in_debug) PARAMS ((
      bfd     *abfd,
      struct internal_syment *sym));
boolean (*_bfd_coff_pointerize_aux_hook) PARAMS ((
      bfd *abfd,
      combined_entry_type *table_base,
      combined_entry_type *symbol,
      unsigned int indaux,
      combined_entry_type *aux));
boolean (*_bfd_coff_print_aux) PARAMS ((
      bfd *abfd,
      FILE *file,
      combined_entry_type *table_base,
      combined_entry_type *symbol,
      combined_entry_type *aux,
      unsigned int indaux));
void (*_bfd_coff_reloc16_extra_cases) PARAMS ((
      bfd     *abfd,
      struct bfd_link_info *link_info,
      struct bfd_link_order *link_order,
      arelent *reloc,
      bfd_byte *data,
      unsigned int *src_ptr,
      unsigned int *dst_ptr));
int (*_bfd_coff_reloc16_estimate) PARAMS ((
```

```
          bfd *abfd,
          asection *input_section,
          arelent *r,
          unsigned int shrink,
          struct bfd_link_info *link_info));
    boolean (*_bfd_coff_sym_is_global) PARAMS ((
          bfd *abfd,
          struct internal_syment *));
    void (*_bfd_coff_compute_section_file_positions) PARAMS ((
          bfd *abfd));
    boolean (*_bfd_coff_start_final_link) PARAMS ((
          bfd *output_bfd,
          struct bfd_link_info *info));
    boolean (*_bfd_coff_relocate_section) PARAMS ((
          bfd *output_bfd,
          struct bfd_link_info *info,
          bfd *input_bfd,
          asection *input_section,
          bfd_byte *contents,
          struct internal_reloc *relocs,
          struct internal_syment *syms,
          asection **sections));
    reloc_howto_type *(*_bfd_coff_rtype_to_howto) PARAMS ((
          bfd *abfd,
          asection *sec,
          struct internal_reloc *rel,
          struct coff_link_hash_entry *h,
          struct internal_syment *sym,
          bfd_vma *addendp));
    boolean (*_bfd_coff_adjust_symndx) PARAMS ((
          bfd *obfd,
          struct bfd_link_info *info,
          bfd *ibfd,
          asection *sec,
          struct internal_reloc *reloc,
          boolean *adjustedp));
    boolean (*_bfd_coff_link_add_one_symbol) PARAMS ((
          struct bfd_link_info *info,
          bfd *abfd,
          const char *name,
          flagword flags,
          asection *section,
          bfd_vma value,
          const char *string,
          boolean copy,
          boolean collect,
          struct bfd_link_hash_entry **hashp));
```

```
} bfd_coff_backend_data;

#define coff_backend_info(abfd) ((bfd_coff_backend_data *) (abfd)->xvec->backend_data)

#define bfd_coff_swap_aux_in(a,e,t,c,ind,num,i) \
        ((coff_backend_info (a)->_bfd_coff_swap_aux_in) (a,e,t,c,ind,num,i))

#define bfd_coff_swap_sym_in(a,e,i) \
        ((coff_backend_info (a)->_bfd_coff_swap_sym_in) (a,e,i))

#define bfd_coff_swap_lineno_in(a,e,i) \
        ((coff_backend_info ( a)->_bfd_coff_swap_lineno_in) (a,e,i))

#define bfd_coff_swap_reloc_out(abfd, i, o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_reloc_out) (abfd, i, o))

#define bfd_coff_swap_lineno_out(abfd, i, o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_lineno_out) (abfd, i, o))

#define bfd_coff_swap_aux_out(a,i,t,c,ind,num,o) \
        ((coff_backend_info (a)->_bfd_coff_swap_aux_out) (a,i,t,c,ind,num,o))

#define bfd_coff_swap_sym_out(abfd, i,o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_sym_out) (abfd, i, o))

#define bfd_coff_swap_scnhdr_out(abfd, i,o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_scnhdr_out) (abfd, i, o))

#define bfd_coff_swap_filehdr_out(abfd, i,o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_filehdr_out) (abfd, i, o))

#define bfd_coff_swap_aouthdr_out(abfd, i,o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_aouthdr_out) (abfd, i, o))

#define bfd_coff_filhsz(abfd) (coff_backend_info (abfd)->_bfd_filhsz)
#define bfd_coff_aoutsz(abfd) (coff_backend_info (abfd)->_bfd_aoutsz)
#define bfd_coff_scnhsz(abfd) (coff_backend_info (abfd)->_bfd_scnhsz)
#define bfd_coff_symesz(abfd) (coff_backend_info (abfd)->_bfd_symesz)
#define bfd_coff_auxesz(abfd) (coff_backend_info (abfd)->_bfd_auxesz)
#define bfd_coff_relsz(abfd)  (coff_backend_info (abfd)->_bfd_relsz)
#define bfd_coff_linesz(abfd) (coff_backend_info (abfd)->_bfd_linesz)
#define bfd_coff_long_filenames(abfd) (coff_backend_info (abfd)->_bfd_coff_long_filenames)
#define bfd_coff_swap_filehdr_in(abfd, i,o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_filehdr_in) (abfd, i, o))

#define bfd_coff_swap_aouthdr_in(abfd, i,o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_aouthdr_in) (abfd, i, o))
```

```
#define bfd_coff_swap_scnhdr_in(abfd, i,o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_scnhdr_in) (abfd, i, o))

#define bfd_coff_swap_reloc_in(abfd, i, o) \
        ((coff_backend_info (abfd)->_bfd_coff_swap_reloc_in) (abfd, i, o))

#define bfd_coff_bad_format_hook(abfd, filehdr) \
        ((coff_backend_info (abfd)->_bfd_coff_bad_format_hook) (abfd, filehdr))

#define bfd_coff_set_arch_mach_hook(abfd, filehdr)\
        ((coff_backend_info (abfd)->_bfd_coff_set_arch_mach_hook) (abfd, filehdr))
#define bfd_coff_mkobject_hook(abfd, filehdr, aouthdr)\
        ((coff_backend_info (abfd)->_bfd_coff_mkobject_hook) (abfd, filehdr, aouthdr))

#define bfd_coff_styp_to_sec_flags_hook(abfd, scnhdr, name)\
        ((coff_backend_info (abfd)->_bfd_styp_to_sec_flags_hook) (abfd, scnhdr, name))

#define bfd_coff_set_alignment_hook(abfd, sec, scnhdr)\
        ((coff_backend_info (abfd)->_bfd_set_alignment_hook) (abfd, sec, scnhdr))

#define bfd_coff_slurp_symbol_table(abfd)\
        ((coff_backend_info (abfd)->_bfd_coff_slurp_symbol_table) (abfd))

#define bfd_coff_symname_in_debug(abfd, sym)\
        ((coff_backend_info (abfd)->_bfd_coff_symname_in_debug) (abfd, sym))

#define bfd_coff_print_aux(abfd, file, base, symbol, aux, indaux)\
        ((coff_backend_info (abfd)->_bfd_coff_print_aux)\
         (abfd, file, base, symbol, aux, indaux))

#define bfd_coff_reloc16_extra_cases(abfd, link_info, link_order, reloc, data, src_ptr, dst_ptr)\
        ((coff_backend_info (abfd)->_bfd_coff_reloc16_extra_cases)\
         (abfd, link_info, link_order, reloc, data, src_ptr, dst_ptr))

#define bfd_coff_reloc16_estimate(abfd, section, reloc, shrink, link_info)\
        ((coff_backend_info (abfd)->_bfd_coff_reloc16_estimate)\
         (abfd, section, reloc, shrink, link_info))

#define bfd_coff_sym_is_global(abfd, sym)\
        ((coff_backend_info (abfd)->_bfd_coff_sym_is_global)\
         (abfd, sym))

#define bfd_coff_compute_section_file_positions(abfd)\
        ((coff_backend_info (abfd)->_bfd_coff_compute_section_file_positions)\
         (abfd))

#define bfd_coff_start_final_link(obfd, info)\
        ((coff_backend_info (obfd)->_bfd_coff_start_final_link)\
```

```
          (obfd, info))
#define bfd_coff_relocate_section(obfd,info,ibfd,o,con,rel,isyms,secs)\
        ((coff_backend_info (ibfd)->_bfd_coff_relocate_section)\
         (obfd, info, ibfd, o, con, rel, isyms, secs))
#define bfd_coff_rtype_to_howto(abfd, sec, rel, h, sym, addendp)\
        ((coff_backend_info (abfd)->_bfd_coff_rtype_to_howto)\
         (abfd, sec, rel, h, sym, addendp))
#define bfd_coff_adjust_symndx(obfd, info, ibfd, sec, rel, adjustedp)\
        ((coff_backend_info (abfd)->_bfd_coff_adjust_symndx)\
         (obfd, info, ibfd, sec, rel, adjustedp))
#define bfd_coff_link_add_one_symbol(info,abfd,name,flags,section,value,string,cp,coll,hashp)\
        ((coff_backend_info (abfd)->_bfd_coff_link_add_one_symbol)\
         (info, abfd, name, flags, section, value, string, cp, coll, hashp))
```

3.2.2.7 Writing relocations

To write relocations, the back end steps 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; 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.

3.2.2.8 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 linenumber 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 ?

3.2.2.9 Reading relocations

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

- Read the entire coff relocation table into memory.
- Process each relocation in turn; first swap it from the external to the internal form.
- Turn the symbol referenced in the relocation's symbol index into a pointer into the canonical symbol table. This table is the same as the one returned by a call to `bfd_canonicalize_symtab`. The back end will call that 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.

3.3 ELF backends

BFD support for ELF formats is being worked on. Currently, the best supported back ends are for sparc and i386 (running svr4 or Solaris 2). Documentation of the internals of the support code still needs to be written. The code is changing quickly enough that we haven't bothered yet.

3.3.0.1 `bfd_elf_find_section`

Synopsis

```
struct elf_internal_shdr *bfd_elf_find_section (bfd *abfd, char *name);
```

Description

Helper functions for GDB to locate the string tables. Since BFD hides string tables from callers, GDB needs to use an internal hook to find them. Sun's .stabstr, in particular, isn't even pointed to by the .stab section, so ordinary mechanisms wouldn't work to find it, even if we had some.

3.4 amiga back end

This section describes the overall structure of the Amiga BFD back end. The linker stuff can be found in See Section 3.6 [amigalink], page 95.

3.5 implementation

The need for a port of the bfd library for Amiga style object (hunk) files arose by the desire to port the GNU debugger gdb to the Amiga. Also, the linker ld should be updated to the current version (2.5.2).

This port bases on the work done by Leonard Norrgard, who started porting gdb. Raphael Luebbert, who supports the ixemul.library, has also worked on implementing the needed `ptrace()` system call and gas2.5.

3.5.1 not supported

Currently, the implementation does not support Amiga link library files, like e.g. amiga.lib. This may be added in a later version, if anyone starts work on it, or I find some time for it. The handling of the symbols in hunk files is a little bit broken:

- The symbols in a load file are totally ignored at the moment, so gdb and gprof do not work.
- The symbols of a object module (Hunk file, starting with HUNK_UNIT) are read in correctly, but HUNK_SYMBOL hunks are also ignored.

The reason for this is the following: Amiga symbol hunks do not allow for much information. Only a name and a value are allowed. On the other hand, a.out format carries along much more information (see, e.g. the entry on set symbols in the ld manual). The old linker copied this information into a HUNK_DEBUG hunk. Now there is the choice:

- ignoring the debug hunk, read in only HUNK_SYMBOL definitions => extra info is lost.
- read in the debug hunk and use the information therein => How can clashs between the information in the debug hunk and HUNK_SYMBOL or HUNK_EXT hunks be avoided ? I haven't decided yet, what to do about this.

Although bfd allows to link together object modules of different flavours, producing a.out style executables does not work on Amiga :-) It should, however, be possible to create a.out files with the -r option of ld (incremental link).

3.5.2 Does it work ?

Currently, the following utillities work:

- objdump
- objcopy
- strip
- nm
- ar
- gas

3.5.3 TODO

- fix fixme:s

* BFD:

- add flag to say if the format allows multiple sections with the same name. Fix bfd_get_section_by_name() and bfd_make_section() accordingly.
- dumpobj.c: the disassembler: use relocation record data to find symbolic names of addresses, when available. Needs new routine where one can specify the source section of the symbol to be printed as well as some rewrite of the disassemble functions.

3.6 amigalink

This is the description of the linker routines for the amiga. In fact, this includes a description of the changes made to the a.out code, in order to build a working linker for the Amiga.

3.6.1 alterations

The file 'aout-amiga.c' defines the amiga a.out backend. It differs from the sun3 backend only in these details:

- The final_link routine is amiga_final_link.
- The routine to get the relocated section contents is aout_bfd_get_relocated_section_contents.

This ensures that the link is performed properly, but has the side effect of loosing performance. The amiga bfd code uses the same amiga_final_link routine, but with a different get_relocated_section_contents entry: amiga_bfd_get_relocated_section_contents. The latter differs from the routine of the a.out backend only in the application of relocs to the section contents.
The usage of a special linker code has one reason: The bfd library assumes that a program is always loaded at a known memory address. This is not a case on an Amiga. So the Amiga format has to take over some relocs to an executable output file. This is not the case with a.out formats, so there relocations can be applied at link time, not at run time, like on the Amiga. The special routines compensate this: instead of applying the relocations, they are copied to the output file, if neccessary. As as consequence, the final_link and get_relocated_section_contents are nearly identical to the original routines from 'linker.c' and 'reloc.c'.

Index

_

A

B

C

H

I

L

S

T

W

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