CTR••–••

libbfd — DRAFT

cygnus reports

The Binary File Descriptor Library

First Edition—`bfd` version < 2.0
April 1991

Steve Chamberlain
Cygnus Support

1 August 1991

Cygnus Support
steve@cygnus.com
BFD, Revision: 1.5
TEXinfo 2.50, Cygnus TR Revision: 1.6

1 Introduction

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

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

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

1.1 History

One spur behind `bfd` was the Intel Oregon's GNU 960 team desire for interoperability of applications on their COFF and b.out file formats. Cygnus was providing GNU support for the team, and Cygnus were contracted to provid the required functionality.

The name came from a conversation Gumby Wallace was having with Richard Stallman about the library, RMS said that it would be quite hard, Gumby 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 Steve Chamberlain (steve@cygnus.com), John Gilmore (gnu@cygnus.com), K. Richard Pixley (rich@cygnus.com) and Gumby Wallace (gumby@cygnus.com) at Cygnus Support in Palo Alto, California.

1.2 How It Works

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

`bfd` provides a common interface to the parts of an object file to 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 structure described in `include/bfd.h`, called `bfd`. Conventionally this pointer is called a `bfd`, and instances of it within code are called `abfd`. All operations on the target object file are applied as methods to the `bfd`, the mapping is defined within `bfd.h` in a set of macros, all beginning `bfd_`something.

For example, this sequence would do what you expect:

```
#include "bfd.h"

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

The metaphor used within `bfd` is that an object file has a header, a numbber of sections containing raw data, a set of relocations and some symbol information. Also, `bfd`s opened upon archives have the additional attribute of an index and contained sub bfds. This approach is find for a.out and coff, but looses efficiency when applied to formats such as S-records and IEEE-695.

1.3 What BFD Version 1 Can't Do

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

1.3.1 Information Loss

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

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

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

This limitation is only a problem when using the linker to read one format and write 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 linker. At the same time, the back end saves away any information which may otherwise be lost. If the data is then written back in the same format, the back end routine will be able to use the canonical form provided by the BFD core as well as the information it prepared earlier. Since there is a great deal of commonality between back ends, this mechanism is very useful. There is no information lost for this reason when linking big endian COFF to little endian COFF, or from `a.out` to `b.out`. When a mixture of formats is linked, the information is only lost from the files whose format differs from the destination.

1.3.2 Mechanism

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

files

Information on target machine architecture, particular implementation and format type are stored on a per-file basis. Other information includes a demand pageable bit and a write protected bit. Note that information like Unix magic numbers is not stored here—only the magic numbers' meaning, so a `ZMAGIC` file would have both the demand pageable bit and the write protected text bit set.

The byte order of the target is stored on a per-file basis, so that big- and little-endian object files may be linked with one another.

sections

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

symbols

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

Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out` type information is stored in the symbol table as long symbol names. This information would be useless to most COFF debuggers and may be thrown away with appropriate command line switches. (The GNU debugger `gdb` does support `a.out` style debugging information in COFF).

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

relocation level

Each canonical BFD relocation record contains a pointer to the symbol to relocate to, the offset of the data to relocate, the section the data is in and a pointer to a relocation type descriptor. Relocation is performed effectively by message passing through the relocation type descriptor and symbol pointer. It allows relocations to be performed on output data using a relocation method only available in one of the input formats. For instance, Oasys provides a byte relocation format. A relocation record requesting this relocation type would point indirectly to a routine to perform this, so the relocation may be performed on a byte being written to a COFF file, even though 68k COFF has no such relocation type.

line numbers

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

What is a backend

2 BFD front end

2.1 typedef bfd

Pointers to bfd structs are the cornerstone of any application using libbfd. References though the bfd and to data in the bfd give the entire bfd functionality.

Finally! The BFD struct itself. This contains the major data about the file, and contains pointers to the rest of the data.

```
struct _bfd
{
```

The filename the application opened the bfd with.

```
  CONST char *filename;
```

A pointer to the target jump table.

```
  struct bfd_target *xvec;
```

*To avoid dragging too many header files into every file that includes bfd.h, IOSTREAM has been declared as a "char *", and MTIME as a "long". Their correct types, to which they are cast when used, are "FILE *" and "time_t".*

The iostream is the result of an fopen on the filename.

```
  char *iostream;
```

Is the file being cached See Chapter 3 [File Caching], page 45.

```
  boolean cacheable;
```

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

```
  boolean target_defaulted;
```

The caching routines use these to maintain an LRU list of bfds.

```
  struct _bfd *lru_prev, *lru_next;
```

When a file is closed by the caching routines, it retains the state here:

```
  file_ptr where;
```

and here:

```
  boolean opened_once;
  boolean mtime_set;
```

File modified time

```
  long mtime;
```

For output files, channel we locked (is this used?).

```
int ifd;
```

The format which belongs to the bfd.

```
  bfd_format format;
```

The direction the bfd was opened with

```
  enum bfd_direction {no_direction = 0,
                      read_direction = 1,
                      write_direction = 2,
```

```
                        both_direction = 3} direction;
```

Format_specific flags

```
      flagword flags;
```

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

```
      file_ptr origin;
```

Remember when output has begun, to stop strange things happening.

```
      boolean output_has_begun;
```

Pointer to linked list of sections

```
      struct sec  *sections;
```

The number of sections

```
      unsigned int section_count;
```

Stuff only usefull for object files: The start address.

```
      bfd_vma start_address;
```

Used for input and output

```
      unsigned int symcount;
```

Symtab for output bfd

```
      struct symbol_cache_entry  **outsymbols;
```

Architecture of object machine, eg m68k

```
      enum bfd_architecture obj_arch;
```

Particular machine within arch, e.g. 68010

```
      unsigned long obj_machine;
```

Stuff only usefull for archives:

```
      PTR arelt_data;
      struct _bfd *my_archive;
      struct _bfd *next;
      struct _bfd *archive_head;
      boolean has_armap;
```

Used by the back end to hold private data.

```
      PTR tdata;
```

Used by the application to hold private data

```
      PTR usrdata;
```

Where all the allocated stuff under this BFD goes

```
      struct obstack memory;
    };
```

2.1.0.1 bfd_set_start_address

Marks the entry point of an output bfd. Returns `true` on success, `false` otherwise.

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

2.1.0.2 bfd_get_mtime

Return cached file modification time (e.g. as read from archive header for archive members, or from file system if we have been called before); else determine modify time, cache it, and return it.

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

2.1.0.3 stuff

```


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

#define bfd_find_nearest_line(abfd, section, symbols, off-
set, filename_ptr, func, line_ptr) \
     BFD_SEND (abfd, _bfd_find_nearest_line,  (abfd, sec-
tion, symbols, offset, filename_ptr, func, line_ptr))

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

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

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

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

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

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

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

2.2 Memory Usage

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

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

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

2.3 Sections

Sections are supported in bfd in `section.c`.

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

2.3.1 Section Input

When a bfd is opened for reading, the section structures are created and attatched to the bfd.

Each section has a name which describes the section in the outside world - for example, `a.out` would contain at least three sections, called `.text`, `.data` and `.bss`.

Sometimes a bfd will contain more than the 'natural' number of sections. A back end may attatch other sections containing constructor data, or an application may add a section (using bfd_make_section) to the sections attatched to an already open bfd. For example, the linker creates a supernumary section `COMMON` for each input file's bfd to hold information about common storage.

The raw data is not necessarily read in at the same time as the section descriptor is created. Some targets may leave the data in place until a `bfd_get_section_contents` call is made. Other back ends may read in all the data at once - For example; an S-record file has to be read once to determine the size of the data. An IEEE-695 file doesn't contain raw data in sections, but data and relocation expressions intermixed, so the data area has to be parsed to get out the data and relocations.

2.3.2 Section Output

To write a new object style bfd, the various sections to be written have to be created. They are attatched to the bfd in the same way as input sections, data is written to the sections using `bfd_set_section_contents`.

The linker uses the fields `output_section` and `output_offset` to create an output file.

The data to be written comes from input sections attatched to the output sections. The output section structure can be considered a filter for the input section, the output section determines the vma of the output data and the name, but the input section determines the offset into the output section of the data to be written.

Eg to create a section "O", starting at 0x100, 0x123 long, containing two subsections, "A" at offset 0x0 (ie at vma 0x100) and "B" at offset 0x20 (ie at vma 0x120) the structures would look like:

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

2.3.3 typedef asection

The shape of a section struct:

```
typedef struct sec {
```

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

```
        CONST char *name;
```

The next section in the list belonging to the bfd, or NULL.

```
        struct sec *next;
```

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

```
    flagword flags;
```

```
   #define SEC_NO_FLAGS   0x000
```

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

```
   #define SEC_ALLOC      0x001
```

Tells the OS to load the section from the file when loading. This would be clear for a .bss section

```
   #define SEC_LOAD       0x002
```

The section contains data still to be relocated, so there will be some relocation information too.

```
   #define SEC_RELOC      0x004
```

Obsolete ?

```
   #define SEC_BALIGN     0x008
```

A signal to the OS that the section contains read only data.

```
   #define SEC_READONLY   0x010
```

The section contains code only.

```
   #define SEC_CODE       0x020
```

The section contains data only.

```
   #define SEC_DATA        0x040
```

The section will reside in ROM.

```
   #define SEC_ROM        0x080
```

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

```
   #define SEC_CONSTRUCTOR 0x100
```

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

```
   #define SEC_CONSTRUCTOR_TEXT 0x1100
   #define SEC_CONSTRUCTOR_DATA 0x2100
   #define SEC_CONSTRUCTOR_BSS  0x3100
```

The section has contents - a bss section could be `SEC_ALLOC | SEC_HAS_CONTENTS`*, a debug section could be* `SEC_HAS_CONTENTS`

```
   #define SEC_HAS_CONTENTS 0x200
```

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

```
   #define SEC_NEVER_LOAD 0x400
```

The base address of the section in the address space of the target.

```
      bfd_vma vma;
```

The size of the section in bytes of the loaded section. This contains a value even if the section has no contents (eg, the size of `.bss`*).*

```
      bfd_size_type size;
```

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

```
        bfd_vma output_offset;
```

The output section through which to map on output.

```
        struct sec *output_section;
```

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

```
        unsigned int alignment_power;
```

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

```
        struct reloc_cache_entry *relocation;
```

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

```
        struct reloc_cache_entry **orelocation;
```

The number of relocation records in one of the above

```
        unsigned reloc_count;
```

Which section is it 0..nth

```
        int index;
```

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

File position of section data

```
        file_ptr filepos;
```

File position of relocation info

```
        file_ptr rel_filepos;
```

File position of line data

```
        file_ptr line_filepos;
```

Pointer to data for applications

```
        PTR userdata;
        struct lang_output_section *otheruserdata;
```

Attached line number information

```
        alent *lineno;
```

Number of line number records

```
        unsigned int lineno_count;
```

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

```
        file_ptr moving_line_filepos;
```

what the section number is in the target world

```
        unsigned 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;
    } asection ;
```

2.3.4 section prototypes

2.3.4.1 bfd_get_section_by_name

Runs through the provided *abfd* and returns the `asection` who's name matches that provided, otherwise NULL. See Section 2.3 [Sections], page 8, for more information.

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

2.3.4.2 bfd_make_section

This function creates a new empty section called *name* and attatches it to the end of the chain of sections for *bfd*. An attempt to create a section with a name which is already in use, returns the old section by that name instead.

Possible errors are:

`invalid_operation`
: If output has already started for this bfd.

`no_memory`
: If obstack alloc fails.

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

2.3.4.3 bfd_set_section_flags

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

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

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

2.3.4.4 bfd_map_over_sections

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

```
  func(abfd, the_section, obj);
PROTO(void, bfd_map_over_sections,
           (bfd *abfd, void (*func)(), PTR 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, ...)
```

2.3.4.5 bfd_set_section_size

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

Possible error returns:

`invalid_operation`
: Writing has started to the bfd, so setting the size is invalid

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

2.3.4.6 bfd_set_section_contents

Sets the contents of the section *section* in bfd *abfd* to the data starting in memory at *data*. The data is written to the output section starting at offset *offset* for *count* bytes.

Normally `true` is returned, else `false`. Possible error returns are:

`no_contents`
: The output section does not have the `SEC_HAS_CONTENTS` attribute, so nothing can be written to it.

`and some more too`

This routine is front end to the back end function `_bfd_set_section_contents`.

```
PROTO(boolean, bfd_set_section_contents,
         (bfd *abfd,
         asection *section,
         PTR data,
         file_ptr offset,
         bfd_size_type count));
```

2.3.4.7 bfd_get_section_contents

This function reads data from *section* in bfd *abfd* into memory starting at *location*. The data is read at an offset of *offset* from the start of the input section, and is read for *count* bytes.

If the contents of a constuctor with the `SEC_CONSTUCTOR` flag set are requested, then the *location* is filled with zeroes.

If no errors occur, `true` is returned, else `false`. Possible errors are:

`unknown yet`

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

2.4 Symbols

bfd trys to maintain as much symbol information as it can when it moves information from file to file. *bfd* passes information to applications though the `asymbol` structure. When the application requests the symbol table, BFD reads the table in the native form and translates parts of it into the internal format. To maintain more than the infomation passed to applications some targets keep some information 'behind the sceans', in a structure only the particular back end knows about. For example, the coff back end keeps the original symbol table structure as well as the canonical structure when a *bfd* is read in. On output, the coff back end can reconstruct the output symbol table so that no information is lost, even information unique to coff which *bfd* doesn't know or understand. If a coff symbol table was read, but was written through an a.out back end, all the coff specific information would be lost. (.. until BFD 2 :).

The symbol table of a bfd is not necessarily read in until a canonicalize request is made. Then the bfd back end fills in a table provided by the application with pointers to the canonical information.

To output symbols, the application provides BFD with a table of pointers to pointers to `asymbols`. This allows applications like the linker to output a symbol as read, since the 'behind the sceens' information will be still available.

2.4.1 Reading Symbols

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

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

storage_needed = get_symtab_upper_bound (abfd);

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

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

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

2.4.2 Writing Symbols

Writing of a symbol table is automatic when a bfd open for writing is closed. The application attatches a vector of pointers to pointers to symbols to the bfd being written, and fills in the symbol count. The close and cleanup code reads through the table provided and performs all the necessary operations. The outputing code must always be provided with an 'owned' symbol; one which has come from another bfd, or one which has been created using `bfd_make_empty_symbol`.

An example showing the creation of a symbol table with only one element:

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

  abfd = bfd_openw("foo","a.out-sunos-big");
  bfd_set_format(abfd, bfd_object);
  new = bfd_make_empty_symbol(abfd);
  new->name = "dummy_symbol";
  new->section = (asection *)0;
  new->flags = BSF_ABSOLUTE | 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.

2.4.3 typedef asymbol

An `asymbol` has the form:

```
typedef struct symbol_cache_entry
{
```

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

```
  struct _bfd *the_bfd;
```

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

```
   CONST char *name;
```

The value of the symbol.

```
   symvalue value;
```

Attributes of a symbol:

```
#define BSF_NO_FLAGS    0x00
```

The symbol has local scope; `static` *in* `C`*. The value is the offset into the section of the data.*

```
#define BSF_LOCAL 0x01
```

The symbol has global scope; initialized data in `C`*. The value is the offset into the section of the data.*

```
#define BSF_GLOBAL 0x02
```

Obsolete

```
#define BSF_IMPORT 0x04
```

The symbol has global scope, and is exported. The value is the offset into the section of the data.

```
#define BSF_EXPORT 0x08
```

The symbol is undefined. `extern` *in* `C`*. The value has no meaning.*

```
#define BSF_UNDEFINED 0x10
```

The symbol is common, initialized to zero; default in `C`*. The value is the size of the object in bytes.*

```
#define BSF_FORT_COMM 0x20
```

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

The symbol is a debugging record. The value has an arbitary meaning.

```
#define BSF_DEBUGGING 0x40
```

The symbol has no section attached, any value is the actual value and is not a relative offset to a section.

```
#define BSF_ABSOLUTE 0x80
```

Used by the linker

```
#define BSF_KEEP        0x10000
#define BSF_WARNING     0x20000
#define BSF_KEEP_G      0x80000
```

Unused

```
#define BSF_WEAK        0x100000
#define BSF_CTOR        0x200000
#define BSF_FAKE        0x400000
```

The symbol used to be a common symbol, but now it is allocated.

```
#define BSF_OLD_COMMON  0x800000
```

The default value for common data.

```
#define BFD_FORT_COMM_DEFAULT_VALUE 0
```

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

```
#define BSF_NOT_AT_END    0x40000
```

Signal that the symbol is the label of constructor section.

```
#define BSF_CONSTRUCTOR   0x1000000
  flagword flags;
```

Aointer to the section to which this symbol is relative, or 0 if the symbol is absolute or undefined. Note that it is not sufficient to set this *location to 0 to mark a symbol as absolute - the flag* `BSF_ABSOLUTE` *must be set also.*

```
      struct sec *section;
```

Back end special data. This is being phased out in favour of making this a union.

```
      PTR udata;
    } asymbol;
```

2.4.4 Symbol Handling Functions

2.4.4.1 get_symtab_upper_bound

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

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

2.4.4.2 bfd_canonicalize_symtab

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

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

2.4.4.3 bfd_set_symtab

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

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

2.4.4.4 bfd_print_symbol_vandf

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

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

2.4.4.5 bfd_make_empty_symbol

This function creates a new `asymbol` structure for the bfd, and returns a pointer to it.

This routine is necessary, since each back end has private information surrounding the `asymbol`. Building your own `asymbol` and pointing to it will not create the private information, and will cause problems later on.

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

2.5 Archives

Gumby, you promised to write this bit...

Archives are supported in bfd in `archive.c`.

An archive is represented internally just like another bfd, with a pointer to a chain of contained bfds. Archives can be created by opening bfds, linking them together and attatching them as children to another bfd and then closing the parent bfd.

2.5.0.1 bfd_get_next_mapent

What this does

```
PROTO(symindex, bfd_get_next_mapent, (bfd *, symindex, carsym **));
```

2.5.0.2 bfd_set_archive_head

Used whilst processing archives. Sets the head of the chain of bfds contained in an archive to *new_head*. (see chapter on archives)

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

2.5.0.3 bfd_get_elt_at_index

```
PROTO(bfd *, bfd_get_elt_at_index, (bfd *, int));
```

2.5.0.4 bfd_openr_next_archived_file

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

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

2.6 File Formats

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

`bfd_object`
: The bfd may contain data, symbols, relocations and debug info.

`bfd_archive`
: The bfd contains other bfds and an optional index.

`bfd_core`
: The bfd contains the result of an executable core dump.

2.6.0.1 bfd_check_format

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

If the bfd has been set to a specific *target* before the call, only the named target and format combination will be checked. If the target has not been set, or has been set to `default` then all the known target backends will be interrogated to determine a match.

The function returns `true` on success, otherwise `false` with one of the following error codes:

> invalid_operation if `format` is not one of `bfd_object`, `bfd_archive` or `bfd_core`.

`system_call_error`
: if an error occured during a read - even some file mismatches can cause system_call_erros

`file_not_recognised`
: none of the backends recognised the file format

`file_ambiguously_recognized`
: more than one backend recognised the file format.

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

2.6.0.2 bfd_set_format

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

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

2.6.0.3 bfd_format_string

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

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

2.7 Relocations

Bfd maintains relocations in much the same was as it maintains symbols; they are left alone until required, then read in en-mass and traslated into an internal form. There is a common routine `bfd_perform_relocation` which acts upon the canonical form to to the actual fixup.

Note that relocations are maintained on a per section basis, whilst symbols are maintained on a per bfd basis.

All a back end has to do to fit the bfd interface is to create as many `struct reloc_cache_entry` as there are relocations in a particuar section, and fill in the right bits:

2.7.0.1 bfd_perform_relocation

The relocation routine returns as a status an enumerated type:

```
   typedef enum bfd_reloc_status {
```

No errors detected

```
     bfd_reloc_ok,
```

The relocation was performed, but there was an overflow.

```
     bfd_reloc_overflow,
```

The address to relocate was not within the section supplied

```
     bfd_reloc_outofrange,
```

Used by special functions

```
     bfd_reloc_continue,
```

Unused

```
     bfd_reloc_notsupported,
```

Unsupported relocation size requested.

```
     bfd_reloc_other,
```

The symbol to relocate against was undefined.

```
     bfd_reloc_undefined,
```

The relocaction was performed, but may not be ok - presently generated only when linking i960 coff files with i960 b.out symbols.

```
     bfd_reloc_dangerous
      }
    bfd_reloc_status_enum_type;
   typedef struct reloc_cache_entry
   {
```

A pointer into the canonical table of pointers

```
     struct symbol_cache_entry **sym_ptr_ptr;
```

offset in section

```
     rawdata_offset address;
```

addend for relocation value

```
     bfd_vma addend;
```

if sym is null this is the section

```
     struct sec *section;
```

Pointer to how to perform the required relocation

```
      CONST struct reloc_howto_struct *howto;
    } arelent;
```

`sym_ptr_ptr`
: The symbol table pointer points to a pointer to the symbol ascociated with the relocation request. This would naturaly be the pointer into the table returned by the back end's get_symtab action. See Section 2.4 [Symbols], page 14. The symbol is referenced through a pointer to a pointer so that tools like the linker can fixup 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 attatched 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. It's interpretation is dependent upon the howto. For example, on the 68k the code:

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

Could be compiled into:

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

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

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE      VALUE
00000006 32        _foo

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

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

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

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

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE      VALUE
00000002 HVRT16    _foo+0x12340000
00000006 LVRT16    _foo+0x12340000

00000000 5da05678           ; or.u r13,r0,0x5678
00000004 1c4d5678           ; ld.b r2,r13,0x5678
00000008 f400c001           ; jmp r1
```

The relocation routine digs out the value from the data, adds it to the addend to get the original offset and then adds the value of _foo. Note that all 32 bits have to be kept around somewhere, to cope with carry from bit 15 to bit 16.

On further example is the sparc and the a.out format. The sparc has a similar problem to the 88k, in that some instructions don't have room for an entire offset, but on the sparc the parts are created odd sized lumps. The designers of the a.out format chose not to use the data within the section for storing part of the offset; all the offset is kept within the reloc. Any thing in the data should be ignored.

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

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

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE      VALUE
00000004 HI22      _foo+0x12345678
00000008 L010      _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
```

`section`

The section field is only used when the symbol pointer field is null. It supplies the section into which the data should be relocated. The field's main use comes from assemblers which do most of the symbol fixups themselves; an assembler may take an internal reference to a label, but since it knows where the label is, it can turn the relocation request from a symbol lookup into a section relative relocation - the relocation emitted has no symbol, just a section to relocate against.

I'm not sure what it means when both a symbol pointer an a section pointer are present. Some formats use this sort of mechanism to describe PIC relocations, but bfd can't to that sort of thing yet.

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

2.7.0.2 reloc_howto_type

The `reloc_howto_type` is a structure which contains all the information that bfd needs to know to tie up a back end's data.

```
typedef CONST struct reloc_howto_struct
{
```

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

```
       unsigned int type;
```

The value the final relocation is shifted right by. This drops unwanted data from the relocation.

```
       unsigned int rightshift;
```

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

```
       unsigned int size;
```

Now obsolete

```
       unsigned int bitsize;
```

Notes that the relocation is relative to the location in the data section of the addend. The relocation function will subtract from the relocation value the address of the location being relocated.

```
       boolean pc_relative;
```

Now obsolete

```
       unsigned int bitpos;
```

Now obsolete

```
       boolean absolute;
```

Causes the relocation routine to return an error if overflow is detected when relocating.

```
       boolean complain_on_overflow;
```

If this field is non null, then the supplied function is called rather than the normal function. This allows really strange relocation methods to be accomodated (eg, i960 callj instructions).

```
       bfd_reloc_status_enum_type (*special_function)();
```

The textual name of the relocation type.

```
       char *name;
```

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

```
       boolean partial_inplace;
```

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

```
       bfd_word src_mask;
```

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

```
       bfd_word dst_mask;
```

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

```
      boolean pcrel_offset;
    } reloc_howto_type;
```

2.7.0.3 HOWTO

The HOWTO define is horrible and will go away.

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

2.7.0.4 reloc_chain

```
typedef unsigned char bfd_byte;

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

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

```
    PROTO(bfd_reloc_status_enum_type,
                    bfd_perform_relocation,
                            (bfd * abfd,
                            arelent *reloc_entry,
                            PTR data,
```

```
                          asection *input_section,
                          bfd *output_bfd));
```

2.8 Core files

Buff output this facinating topic

2.8.0.1 bfd_core_file_failing_command

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

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

2.8.0.2 bfd_core_file_failing_signal

Returns the signal number which caused the core dump which generated the file the bfd is attatched to.

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

2.8.0.3 core_file_matches_executable_p

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

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

2.9 Targets

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 operatios performed are:

- First a bfd is created by calling the internal routine `new_bfd`, then `bfd_find_target` is called with the target string supplied to `bfd_openr` and the new bfd pointer.
- If a null target string was provided to `bfd_find_target`, it looks up the environment variable `GNUTARGET` and uses that as the target string.
- If the target string is still NULL, or the target string is `default`, then the first item in the target vector is used as the target type. See ⟨undefined⟩ [targets], page ⟨undefined⟩.
- Otherwise, the elements in the target vector are inspected one by one, until a match on target name is found. When found, that is used.
- Otherwise the error `invalid_target` is returned to `bfd_openr`.
- `bfd_openr` attempts to open the file using `bfd_open_file`, and returns the bfd.

Once the bfd has been opened and the target selected, the file format may be determined. This is done by calling `bfd_check_format` on the bfd with a suggested format. The routine returns `true` when the application guesses right.

2.9.0.1 bfd_target

2.9.1 bfd_target

This structure contains everything that BFD knows about a target. It includes things like its byte order, name, what routines to call to do various operations, etc.

Every BFD points to a target structure with its "xvec" member.

Shortcut for declaring fields which are prototyped function pointers, while avoiding anguish on compilers that don't support protos.

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

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

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

For operations which index on the bfd format

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

This is the struct which defines the type of BFD this is. The "xvec" member of the struct bfd itself points here. Each module that implements access to a different target under BFD, defines one of these.

FIXME, these names should be rationalised with the names of the entry points which call them. Too bad we can't have one macro to define them both!

```
    typedef struct bfd_target
    {
```

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

```
      char *name;
```

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

```
      enum target_flavour_enum {
        bfd_target_aout_flavour_enum,
        bfd_target_coff_flavour_enum,
        bfd_target_ieee_flavour_enum,
        bfd_target_oasys_flavour_enum,
        bfd_target_srec_flavour_enum} flavour;
```

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

```
      boolean byteorder_big_p;
```

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

```
      boolean header_byteorder_big_p;
```

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

```
      flagword object_flags;
```

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

```
      flagword section_flags;
```

The pad character for filenames within an archive header.

```
      char ar_pad_char;
```

The maximum number of characters in an archive header.

```
     unsigned short ar_max_namelen;
```

The minimum alignment restriction for any section.

```
      unsigned int align_power_min;
```

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

```
      SDEF (bfd_vma,      bfd_getx64, (bfd_byte *));
```

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

Byte swapping for the headers

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

Format dependent routines, these turn into vectors of entry points within the target vector structure; one for each format to check.

*Check the format of a file being read. Return bfd_target * or zero.*

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

Set the format of a file being written.

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

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

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

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

Core file entry points

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

Archive entry points

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

Standard stuff.

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

Symbols and reloctions

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

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

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

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

  SDEF (void, _bfd_debug_info_start, (bfd *));
```

```
    SDEF (void, _bfd_debug_info_end, (bfd *));
    SDEF (void, _bfd_debug_info_accumulate, (bfd *, struct
  sec  *));
```

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

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

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

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

  } bfd_target;
```

2.9.1.1 bfd_find_target

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

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

2.9.1.2 bfd_target_list

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

```
  PROTO(CONST char **,bfd_target_list,());
```

2.10 Architectures

BFD's idea of an architecture is implimented in `archures.c`. BFD keeps two atoms in a bfd describing the architecture of the data attatched to the bfd, the `enum bfd_architecture arch` field and the `unsigned long machine` field.

2.10.0.1 bfd_architecture

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

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

#define bfd_mach_i960_core      1
#define bfd_mach_i960_ka_sa     2
#define bfd_mach_i960_kb_sb     3
#define bfd_mach_i960_mc        4
#define bfd_mach_i960_xa        5
#define bfd_mach_i960_ca        6

  bfd_arch_a29k,      /* AMD 29000 */
  bfd_arch_sparc,     /* SPARC */
  bfd_arch_mips,      /* MIPS Rxxxx */
  bfd_arch_i386,      /* Intel 386 */
  bfd_arch_ns32k,     /* National Semiconductor 32xxx */
  bfd_arch_tahoe,     /* CCI/Harris Tahoe */
  bfd_arch_i860,      /* Intel 860 */
  bfd_arch_romp,      /* IBM ROMP RS/6000 */
  bfd_arch_alliant,   /* Alliant */
  bfd_arch_convex,    /* Convex */
  bfd_arch_m88k,      /* Motorola 88xxx */
  bfd_arch_pyramid,   /* Pyramid Technology */
  bfd_arch_h8_300,    /* Hitachi H8/300 */
  bfd_arch_last
  };
```

stuff

2.10.0.2 bfd_prinable_arch_mach

Return a printable string representing the architecture and machine type. The result is only good until the next call to bfd_printable_arch_mach.

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

2.10.0.3 bfd_scan_arch_mach

Scan a string and attempt to turn it into an archive and machine type combination.

```
PROTO(boolean, bfd_scan_arch_mach,
    (CONST char *, enum bfd_architecture *, unsigned
long *));
```

2.10.0.4 bfd_arch_compatible

This routine is used to determine whether two BFDs' architectures and machine types are compatible. It calculates the lowest common denominator between the two architectures and machine types implied by the bfds and sets the objects pointed at by *archp* and *machine* if non NULL.

This routine returns `true` if the bfds are of compatible type, otherwise `false`.

```
PROTO(boolean, bfd_arch_compatible,
     (bfd *abfd,
     bfd *bbfd,
     enum bfd_architecture *archp,
     unsigned long *machinep));
```

2.10.0.5 bfd_set_arch_mach

Set atch mach

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

2.11 Opening and Closing BFDs

2.11.0.1 bfd_openr

Opens the file supplied (using fopen) with the target supplied, it returns a pointer to the created bfd.

If NULL is returned then an error has occured. Possible errors are no_memory, invalid_target or system_call error.

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

2.11.0.2 bfd_fdopenr

bfd_fdopenr is to bfd_fopenr much like fdopen is to fopen. It opens a bfd on a file already described by the *fd* supplied.

Possible errors are no_memory, invalid_target and system_call error.

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

2.11.0.3 bfd_openw

Creates a bfd, associated with file *filename*, using the file format *target*, and returns a pointer to it.

Possible errors are system_call_error, no_memory, invalid_target.

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

2.11.0.4 bfd_close

This function closes a bfd. If the bfd was open for writing, then pending operations are completed and the file written out and closed. If the created file is executable, then `chmod` is called to mark it as such.

All memory attatched to the bfd's obstacks is released.

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

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

2.11.0.5 bfd_create

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

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

2.11.0.6 bfd_alloc_size

Return the number of bytes in the obstacks connected to the supplied bfd.

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

2.11.1.7 bfd_put_size

2.11.1.8 bfd_get_size

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

```
#define bfd_put_8(abfd, val, ptr) \
                (*((char *)ptr) = (char)val)
#define bfd_get_8(abfd, ptr) \
                (*((char *)ptr))
#define bfd_put_16(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx16, (val,ptr))
#define bfd_get_16(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx16, (ptr))
#define bfd_put_32(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx32, (val,ptr))
#define bfd_get_32(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx32, (ptr))
#define bfd_put_64(abfd, val, ptr) \
                BFD_SEND(abfd, bfd_putx64, (val, ptr))
#define bfd_get_64(abfd, ptr) \
                BFD_SEND(abfd, bfd_getx64, (ptr))
```

2.11.1.9 bfd_h_put_size

2.11.1.10 bfd_h_get_size

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

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

2.11.1.11 bfd_log2

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

```
PROTO(bfd_vma, bfd_log2,(bfd_vma x));
```

2.12 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.

2.12.0.1 BFD_CACHE_MAX_OPEN

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

```
#define BFD_CACHE_MAX_OPEN 10
```

2.12.0.2 bfd_last_cache

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.

```
extern bfd *bfd_last_cache;
```

2.12.0.3 bfd_cache_lookup

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

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

2.12.0.4 bfd_cache_init

Initialize a BFD by putting it on the cache LRU.

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

2.12.0.5 bfd_cache_close

Remove the bfd from the cache. If the attatched file is open, then close it too.

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

2.12.0.6 bfd_open_file

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

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

2.12.0.7 bfd_cache_lookup_worker

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

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

3 BFD back end

All of bfd lives in one directory.

3.1 a.out backends

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 and `demo64` for a demonstration of a 64 bit a.out format.

The base file `aoutx.h` defines general mechanisms for reading and writing records to and from disk, and various other methods which bfd requires. It is included by `aout32.c` and `aout64.c` to form the names aout_32_swap_exec_header_in, aout_64_swap_exec_header_in, etc.

As an example, this is what goes on to make the back end for a sun4, from aout32.c

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

Which exports names: ...

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

from sunos.c

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

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

```
sunos_big_vec
```

3.1.1 relocations

The file `aoutx.h` caters for both the *standard* and *extended* forms of a.out relocation records.

The standard records are characterised by containing only an address, a symbol index and a type field. The extended records (used on 29ks and sparcs) also have a full integer for an addend.

3.1.2 Internal Entry Points

`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

Swaps the information in an executable header taken from a raw byte stream memory image, into the internal exec_header structure.

```
PROTO(void, aout_<size>_swap_exec_header_in,
      (bfd *abfd,
      struct external_exec *raw_bytes,
      struct internal_exec *execp));
```

3.1.2.2 aout_<size>_swap_exec_header_out

Swaps the information in an internal exec header structure into the supplied buffer ready for writing to disk.

```
PROTO(void, aout_<size>_swap_exec_header_out,
  (bfd *abfd,
   struct internal_exec *execp,
   struct external_exec *raw_bytes));
```

3.1.2.3 aout_<size>_some_aout_object_p

Some A.OUT variant thinks that the file whose format we're checking is an a.out file. Do some more checking, and set up for access if it really is. Call back to the calling environments "finish up" function just before returning, to handle any last-minute setup.

```
PROTO(bfd_target *, aout_<size>_some_aout_object_p,
 (bfd *abfd,
  bfd_target *(*callback_to_real_object_p)()));
```

3.1.2.4 aout_<size>_mkobject

This routine initializes a bfd for use with a.out files.

```
PROTO(boolean, aout_<size>_mkobject, (bfd *));
```

3.1.2.5 aout_<size>_machine_type

Keep track of machine architecture and machine type for a.out's. Return the machine_type for a particular arch&machine, or M_UNKNOWN if that exact arch&machine can't be represented in a.out format.

If the architecture is understood, machine type 0 (default) should always be understood.

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

3.1.2.6 aout_<size>_set_arch_mach

Sets the architecture and the machine of the bfd to those values supplied. Verifies that the format can support the architecture required.

```
PROTO(boolean, aout_<size>_set_arch_mach,
 (bfd *,
  enum bfd_architecture,
  unsigned long machine));
```

3.1.2.7 aout_<size>new_section_hook

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

```
PROTO(boolean, aout_<size>_new_section_hook,
        (bfd *abfd,
  asection *newsect));
```

3.2 coff backends

BFD supports a number of different flavours of coff format. The major difference between formats are the sizes and alignments of fields in structures on disk, and the occasional extra field.

Coff in all its varieties is implimented with a few common files and a number of implementation specific files. For example, The 88k bcs coff format is implemented in the file `m88k-bcs.c`. This file `#include`s `m88k-bcs.h` which defines the external structure of the coff format for the 88k, and `internalcoff.h` which defines the internal structure. `m88k-bcs.c` also defines the relocations used by the 88k format See Section 2.7 [Relocations], page 28. Then the major portion of coff code is included (`coffcode.h`) which defines the methods used to act upon the types defined in `m88k-bcs.h` and `internalcoff.h`.

The Intel i960 processor version of coff is implemented in `icoff.c`. This file has the same structure as `m88k-bcs.c`, except that it includes `intel-coff.h` rather than `m88k-bcs.h`.

3.2.1 Porting To A New Version of Coff

The recommended method is to select from the existing implimentations the version of coff which is most like the one you want to use, for our purposes, we'll say that i386 coff is the one you select, and that your coff flavour is called foo. Copy the `i386coff.c` to `foocoff.c`, copy `../include/i386coff.h` to `../include/foocoff.h` and add the lines to `targets.c` and `Makefile.in` so that your new back end is used.

Alter the shapes of the structures in `../include/foocoff.h` so that they match what you need. You will probably also have to add `#ifdef`s to the code in `internalcoff.h` and `coffcode.h` if your version of coff is too wild.

You can verify that your new bfd backend works quite simply by building `objdump` from the `binutils` directory, and making sure that its version of what's going on at your host systems idea (assuming it has the pretty standard coff dump utility (usually called `att-dump` or just `dump`)) are the same.

Then clean up your code, and send what you've done to Cygnus. Then your stuff will be in the next release, and you won't have to keep integrating it.

3.2.2 How The Coff Backend Works

3.2.2.1 Bit Twiddling

Each flavour of coff supported in bfd has its own header file descibing the external layout of the structures. There is also an internal description of the coff layout (in `internalcoff.h`) file ( ). A major function of the coff backend is swapping the bytes and twiddling the bits to translate the external form of the structures into the normal internal form. This is all performed in the `bfd_swap`_*thing*_*direction* routines. Some elements are different sizes between different versions of coff, it is the duty of the coff version specific include file to override the definitions of various packing routines in `coffcode.h`. Eg the size of line number entry in coff is sometimes 16 bits, and sometimes 32 bits. `#define`ing `PUT_LNSZ_LNNO` and `GET_LNSZ_LNNO` will select the correct one. No doubt, some day someone will find a version of coff which has a varying field size not catered for at the moment. To port bfd, that person will have to add more `#defines`.

Three of the bit twiddling routines are exported to `gdb`; `coff_swap_aux_in`, `coff_swap_sym_in` and `coff_swap_linno_in`. `GDB` reads the symbol table on its own, but uses bfd to fix things up.

3.2.2.2 Symbol Reading

The simple canonical form for symbols used by bfd is not rich enough to keep all the information available in a coff symbol table. The back end gets around this by keeping the original symbol table around, "behind the sceens".

When a symbol table is requested (through a call to `bfd_canonicalize_symtab`, a request gets through to `get_normalized_symtab`. This reads the symbol table from the coff file and swaps all the structures inside into the internal form. It also fixes up all the pointers in the table (represented in the file by offsets from the first symbol in the table) into physical pointers to elements in the new internal table. This involves some work since the meanings of fields changes depending upon context; a field that is a pointer to another structure in the symbol table at one moment may be the size in bytes of a structure in the next.

Another pass is made over the table. All symbols which mark file names (`C_FILE` symbols) are modified so that the internal string points to the value in the auxent (the real filename) rather than the normal text associated with the symbol (`".file"`).

At this time the symbol names are moved around. Coff stores all symbols less than nine characters long physically within the symbol table, longer strings are kept at the end of the file in the string table. This pass moves all strings into memory, and replaces them with pointers to the strings.

The symbol table is massaged once again, this time to create the canonical table used by the bfd application. Each symbol is inspected in turn, and a decision made (using the `sclass` field) about the various flags to set in the `asymbol` See Section 2.4 [Symbols], page 14. The generated canonical table shares strings with the hidden internal symbol table.

Any linenumbers are read from the coff file too, and attatched to the symbols which own the functions the linenumbers belong to.

3.2.2.3 Symbol Writing

Writing a symbol to a coff file which didn't come from a coff file will lose any debugging information. The `asymbol` structure remembers the bfd from which was born, and on output the back end makes sure that the same destination target as source target is present.

When the symbols have come from a coff file then all the debugging information is preserved.

Symbol tables are provided for writing to the back end in a vector of pointers to pointers. This allows applications like the linker to accumulate and output large symbol tables without having to do too much byte copying.

The symbol table is not output to a writable bfd until it is closed. The order of operations on the canonical symbol table at that point are:

`coff_renumber_symbols`

> This function runs through the provided symbol table and patches each symbol marked as a file place holder (`C_FILE`) to point to the next file place holder in the list. It also marks each `offset` field in the list with the offset from the first symbol of the current symbol.
>
> Another function of this procedure is to turn the canonical value form of bfd into the form used by coff. Internally, bfd expects symbol values to be offsets from a section base; so a symbol physically at 0x120, but in a section starting at 0x100, would have the value 0x20. Coff expects symbols to contain their final value, so symbols have their values changed at this point to reflect their sum with their owning section. Note that this transformation uses the `output_section` field of the `asymbol`'s `asection` See Section 2.3 [Sections], page 8.

`coff_mangle_symbols`

> This routine runs though the provided symbol table and uses the offsets generated by the previous pass and the pointers generated when the symbol table was read in to create the structured hierachy required by coff. It changes each pointer to a symbol to an index into the symbol table of the symbol being referenced.

`coff_write_symbols`

> This routine runs through the symbol table and patches up the symbols from their internal form into the coff way, calls the bit twiddlers and writes out the tabel to the file.

The hidden information for an asymbol is:

```
typedef struct coff_ptr_struct
{
```

Remembers the offset from the first symbol in the file for this symbol. Generated by `coff_renumber_symbols`.

```
  unsigned int offset;
```

Should the tag field of this symbol be renumbered. Created by `coff_pointerize_aux`.

```
  char fix_tag;
```

Should the endidx field of this symbol be renumbered. Created by `coff_pointerize_aux`.

```
  char fix_end;
```

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

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

Each canonical asymbol really looks like this:

```
typedef struct coff_symbol_struct
{
```

The actual symbol which the rest of bfd works with

```
  asymbol symbol;
```

A pointer to the hidden information for this symbol

```
  combined_entry_type *native;
```

A pointer to the linenumber information for this symbol

```
  struct lineno_cache_entry *lineno;
} coff_symbol_type;
```

3.2.2.4 Writing Relocations

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

Then the `internal_reloc` is swapped into the shape of an `external_reloc` and written out to disk.

3.2.2.5 Reading Linenumbers

Createing the linenumber table is done by reading in the entire coff linenumber table, and creating another table for internal use.

A coff line number table is structured so that each function is marked as having a line number of 0. Each line within the function is an offset from the first line in the function. The base of the line number information for the table is stored in the symbol associated with the function.

The information is copied from the external to the internal table, and each symbol which marks a function is marked by pointing its...

How does this work ?

3.2.2.6 Reading Relocations

Coff relocations are easily transformed into the internal bfd form (`arelent`).

Reading a coff relocation table is done in the following stages:

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

Function Index

Index

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