1. Introduction. This is GB_IO, the input/output module used by all GraphBase routines to access data files. It doesn't actually do any output; but somehow 'input/output' sounds like a more useful title than just 'input'.

All files of GraphBase data are designed to produce identical results on almost all existing computers and operating systems. Each line of each file contains at most 79 characters. Each character is either a blank or a digit or an uppercase letter or a lowercase letter or a standard punctuation mark. Blank characters at the end of each line are "invisible"; that is, they have no perceivable effect. Hence identical results will be obtained on record-oriented systems that pad every line with blanks.

The data is carefully sum-checked so that defective input files have little chance of being accepted.

2. Changes might be needed when these routines are ported to different systems. Sections of the program that are most likely to require such changes are listed under 'system dependencies' in the index.

A validation program is provided so that installers can tell if GB_IO is working properly. To make the test, simply run `test_io`.

```
⟨ test_io.c 2 ⟩ ≡
#include "gb_io.h"      /* all users of GB_IO should include this header file */
#define exit_test(m)    /* we invoke this macro if something goes wrong */
  { fprintf(stderr,"%s!\n(Error code = %ld)\n",m,io_errors); return -1; }

int main()
{
  ⟨ Test the gb_open routine; exit if there's trouble 28 ⟩;
  ⟨ Test the sample data lines; exit if there's trouble 27 ⟩;
  ⟨ Test the gb_close routine; exit if there's trouble 38 ⟩;
  printf("OK, the gb_io routines seem to work!\n");
  return 0;
}
```

3. The external variable *io_errors* mentioned in the previous section will be set nonzero if any anomalies are detected. Errors won't occur in normal use of GraphBase programs, so no attempt has been made to provide a user-friendly way to decode the nonzero values that *io_errors* might assume. Information is simply gathered in binary form; system wizards who might need to do a bit of troubleshooting should be able to decode *io_errors* without great pain.

```
#define cant_open_file 1      /* bit set in io_errors if fopen fails */
#define cant_close_file 2     /* bit set if fclose fails */
#define bad_first_line 4      /* bit set if the data file's first line isn't legit */
#define bad_second_line 8     /* bit set if the second line doesn't pass muster */
#define bad_third_line 10     /* bit set if the third line is awry */
#define bad_fourth_line 20    /* guess when this bit is set */
#define file_ended_prematurely 40     /* bit set if fgets fails */
#define missing_newline 80    /* bit set if line is too long or '\n' is missing */
#define wrong_number_of_lines 100     /* bit set if the line count is wrong */
#define wrong_checksum 200    /* bit set if the check sum is wrong */
#define no_file_open 400      /* bit set if user tries to close an unopened file */
#define bad_last_line 800     /* bit set if final line has incorrect form */
```

4. The C code for GB_IO doesn't have a main routine; it's just a bunch of subroutines to be incorporated into programs at a higher level via the system loading routine. Here is the general outline of `gb_io.c`:

⟨ Header files to include 7 ⟩
⟨ Preprocessor definitions ⟩
⟨ External declarations 5 ⟩
⟨ Private declarations 8 ⟩
⟨ Internal functions 9 ⟩
⟨ External functions 12 ⟩

5. Every external variable is declared twice in this `CWEB` file: once for GB_IO itself (the "real" declaration for storage allocation purposes) and once in `gb_io.h` (for cross-references by GB_IO users).

⟨ External declarations 5 ⟩ ≡
```
long io_errors;      /* record of anomalies noted by GB_IO routines */
```
This code is used in section 4.

6. ⟨ `gb_io.h` 6 ⟩ ≡
```
⟨ Header files to include 7 ⟩
extern long io_errors;      /* record of anomalies noted by GB_IO routines */
```
See also sections 13, 16, 19, 21, 23, 25, 29, and 41.

7. We will stick to standard C-type input conventions. We'll also have occasion to use some of the standard string operations.

⟨ Header files to include 7 ⟩ ≡
```
#include <stdio.h>
#ifdef SYSV
#include <string.h>
#else
#include <strings.h>
#endif
```
This code is used in sections 4 and 6.

8. Inputting a line. The GB_IO routines get their input from an array called *buffer*. This array is internal to GB_IO—its contents are hidden from user programs. We make it 81 characters long, since the data is supposed to have at most 79 characters per line, followed by newline and null.

```
⟨ Private declarations 8 ⟩ ≡
  static char buffer[81];      /* the current line of input */
  static char *cur_pos = buffer;      /* the current character of interest */
  static FILE *cur_file;      /* current file, or Λ is none is open */
```

See also sections 10, 11, and 33.

This code is used in section 4.

9. Here's a basic subroutine to fill the *buffer*. The main feature of interest is the removal of trailing blanks. We assume that *cur_file* is open.

Notice that a line of 79 characters (followed by `'\n'`) will just fit into the buffer, and will cause no errors. A line of 80 characters will be split into two lines and the *missing_newline* message will occur, because of the way *fgets* is defined. A *missing_newline* error will also occur if the file ends in the middle of a line, or if a null character (`'\0'`) occurs within a line.

```
⟨ Internal functions 9 ⟩ ≡
  static void fill_buf( )
  { register char *p;
    if (¬fgets(buffer, 81, cur_file)) {
      io_errors |= file_ended_prematurely;
      buffer[0] = more_data = 0;
    }
    for (p = buffer; *p; p++) ;      /* advance to first null character */
    if (p-- ≡ buffer ∨ *p ≠ '\n') {
      io_errors |= missing_newline;
      p++;
    }
    while (--p ≥ buffer ∧ *p ≡ '␣') ;      /* move back over trailing blanks */
    *++p = '\n';
    *++p = 0;      /* newline and null are always present at end of line */
    cur_pos = buffer;      /* get ready to read buffer[0] */
  }
```

See also section 15.

This code is used in section 4.

10. Checksums. Each data file has a "magic number," which is defined to be

$$\left(\sum_l 2^l c_l\right) \bmod p\,.$$

Here p is a large prime number, and c_l denotes the internal code corresponding to the lth-from-last data character read (including newlines but not nulls).

The "internal codes" c_l are computed in a system-independent way: Each character c in the actual encoding scheme being used has a corresponding *icode*, which is the same on all systems. For example, the *icode* of `'0'` is zero, regardless of whether `'0'` is actually represented in ASCII or EBCDIC or some other scheme. (We assume that every modern computer system is capable of printing at least 95 different characters, including a blank space.)

We will accept a data file as error-free if it has the correct number of lines and ends with the proper magic number.

```
⟨ Private declarations 8 ⟩ +≡
  static char icode[256];      /* mapping of characters to internal codes */
  static long checksum_prime = (1L << 30) - 83;      /* large prime such that 2p + 100 won't overflow */
  static long magic;      /* current checksum value */
  static long line_no;      /* current line number in file */
  static long final_magic;      /* desired final magic number */
  static long tot_lines;      /* total number of data lines */
  static char more_data;      /* is there data still waiting to be read? */
```

11. The *icode* mapping is defined by a single string, *imap*, such that character *imap*$[k]$ has *icode* value k. There are 96 characters in *imap*, namely the 94 standard visible ASCII codes plus space and newline. If EBCDIC code is used instead of ASCII, the cents sign ¢ should take the place of single-left-quote `‘`, and ¬ should take the place of `~`.

All characters that don't appear in *imap* are given the same *icode* value, called *unexpected_char*. Such characters should be avoided in GraphBase files whenever possible. (If they do appear, they can still get into a user's data, but we don't distinguish them from each other for checksumming purposes.)

The *icode* table actually plays a dual role, because we've rigged it so that codes 0–15 come from the characters `"0123456789ABCDEF"`. This facilitates conversion of decimal and hexadecimal data. We can also use it for radices higher than 16.

```
#define unexpected_char 127      /* default icode value */
⟨ Private declarations 8 ⟩ +≡
  static char *imap = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw\
      xyz_^~&@,;.:?!%#$+-*/|\\<=>()[]{}‘’\"␣\n";
```

12. Users of GB_IO can look at the *imap*, but they can't change it.

```
⟨ External functions 12 ⟩ ≡
  char imap_chr(d)
       long d;
  {
    return d < 0 ∨ d > strlen(imap) ? '\0' : imap[d];
  }
  long imap_ord(c)
       char c;
  {
    ⟨ Make sure that icode has been initialized 14 ⟩;
    return (c < 0 ∨ c > 255) ? unexpected_char : icode[c];
  }
```

See also sections 17, 18, 20, 22, 24, 26, 30, 32, 39, and 42.

This code is used in section 4.

13.

```
⟨ gb_io.h 6 ⟩ +≡
#define unexpected_char 127
  extern char imap_chr( );   /* the character that maps to a given character */
  extern long imap_ord( );   /* the ordinal number of a given character */
```

14.

```
⟨ Make sure that icode has been initialized 14 ⟩ ≡
  if (¬icode['1']) icode_setup( );
```

This code is used in sections 12 and 30.

15.

```
⟨ Internal functions 9 ⟩ +≡
  static void icode_setup( )
  { register long k;
    register char *p;
    for (k = 0; k < 256; k++) icode[k] = unexpected_char;
    for (p = imap, k = 0; *p; p++, k++) icode[*p] = k;
  }
```

16. Now we're ready to specify some external subroutines that do input. Calling *gb_newline*( ) will read the next line of data into *buffer* and update the magic number accordingly.

```
⟨ gb_io.h 6 ⟩ +≡
  extern void gb_newline( );   /* advance to next line of the data file */
  extern long new_checksum( );   /* compute change in magic number */
```

17. Users can compute checksums as *gb_newline* does, but they can't change the (private) value of *magic*.

```
⟨ External functions 12 ⟩ +≡
  long new_checksum(s, old_checksum)
       char *s;   /* a string */
       long old_checksum;
  { register long a = old_checksum;
    register char *p;
    for (p = s; *p; p++) a = (a + a + icode[*p]) % checksum_prime;
    return a;
  }
```

18. The magic checksum is not affected by lines that begin with `*`.

```
⟨ External functions 12 ⟩ +≡
  void gb_newline( )
  {
    if (++line_no > tot_lines) more_data = 0;
    if (more_data) {
      fill_buf( );
      if (buffer[0] ≠ '*') magic = new_checksum(buffer, magic);
    }
  }
```

19. Another simple routine allows a user to read (but not write) the variable *more_data*.

```
⟨ gb_io.h 6 ⟩ +≡
  extern long gb_eof( );      /* has the data all been read? */
```

20.

```
⟨ External functions 12 ⟩ +≡
  long gb_eof( )
  {
    return ¬more_data;
  }
```

21. Parsing a line. The user can input characters from the buffer in several ways. First, there's a basic *gb_char*( ) routine, which returns a single character. The character is `'\n'` if the last character on the line has already been read (and it continues to be `'\n'` until the user calls *gb_newline*).

The current position in the line, *cur_pos*, always advances when *gb_char* is called, unless *cur_pos* was already at the end of the line. There's also a *gb_backup*( ) routine, which moves *cur_pos* one place to the left unless it was already at the beginning.

```
⟨ gb_io.h 6 ⟩ +≡
  extern char gb_char( );      /* get next character of current line, or '\n' */
  extern void gb_backup( );      /* move back ready to scan a character again */
```

22.
```
⟨ External functions 12 ⟩ +≡
  char gb_char( )
  {
    if (*cur_pos) return (*cur_pos++);
    return '\n';
  }
  void gb_backup( )
  {
    if (cur_pos > buffer) cur_pos--;
  }
```

23. There are two ways to read numerical data. The first, *gb_digit*(d), expects to read a single character in radix d, using *icode* values to specify digits greater than 9. (Thus, for example, `'A'` represents the hexadecimal digit for decimal 10.) If the next character is a valid d-git, *cur_pos* moves to the next character and the numerical value is returned. Otherwise *cur_pos* stays in the same place and -1 is returned.

The second routine, *gb_number*(d), reads characters and forms an unsigned radix-d number until the first non-digit is encountered. The resulting number is returned; it is zero if no digits were found. No errors are possible with this routine, because it uses **unsigned long** arithmetic.

```
⟨ gb_io.h 6 ⟩ +≡
  extern long gb_digit( );      /* gb_digit(d) reads a digit between 0 and d − 1 */
  extern unsigned long gb_number( );      /* gb_number(d) reads a radix-d number */
```

24. The value of d should be at most 127, if users want their programs to be portable, because C does not treat larger **char** values in a well-defined manner. In most applications, d is of course either 10 or 16.

```
⟨ External functions 12 ⟩ +≡
  long gb_digit(d)
       char d;
  {
    if (icode[*cur_pos] < d) return icode[*cur_pos++];
    return −1;
  }
  unsigned long gb_number(d)
       char d;
  { register unsigned long a = 0;
    icode[0] = d;      /* make sure '\0' is a nondigit */
    while (icode[*cur_pos] < d) a = a * d + icode[*cur_pos++];
    return a;
  }
```

25. The final subroutine for fetching data is *gb_string*(p, c), which stores a null-terminated string into locations starting at p. The string starts at *cur_pos* and ends just before the first appearance of character c. If $c \equiv$ `'\n'`, the string will stop at the end of the line. If c doesn't appear in the buffer at or after *cur_pos*, the last character of the string will be the `'\n'` that is always inserted at the end of a line, unless the entire line has already been read. (If the entire line has previously been read, the empty string is always returned.) After the string has been copied, *cur_pos* advances past it.

In order to use this routine safely, the user should first check that there is room to store up to 81 characters beginning at location p. A suitable place to put the result, called *str_buf*, is provided for the user's convenience.

The location following the stored string is returned. Thus, if the stored string has length l (not counting the null character that is stored at the end), the value returned will be $p + l + 1$.

```
⟨ gb_io.h 6 ⟩ +≡
#define STR_BUF_LENGTH 160
  extern char str_buf[];      /* safe place to receive output of gb_string */
  extern char *gb_string();   /* gb_string(p, c) reads a string delimited by c into bytes starting at p */
```

26.

```
#define STR_BUF_LENGTH 160
```

```
⟨ External functions 12 ⟩ +≡
  char str_buf[STR_BUF_LENGTH];   /* users can put strings here if they wish */
  char *gb_string(p, c)
      char *p;    /* where to put the result */
      char c;     /* character following the string */
  {
    while (*cur_pos ∧ *cur_pos ≠ c) *p++ = *cur_pos++;
    *p++ = 0;
    return p;
  }
```

27. Here's how we test those routines in `test_io`: The first line of test data consists of 79 characters, beginning with 64 zeroes and ending with '`123456789ABCDEF`'. The second line is completely blank. The third and final line says '`Oops:(intentional mistake)`'.

```
⟨ Test the sample data lines; exit if there's trouble 27 ⟩ ≡
  if (gb_number(10) ≠ 123456789) io_errors |= 1L ≪ 20;      /* decimal number not working */
  if (gb_digit(16) ≠ 10) io_errors |= 1L ≪ 21;      /* we missed the A following the decimal number */
  gb_backup( ); gb_backup( );      /* get set to read '9A' again */
  if (gb_number(16) ≠ #9ABCDEF) io_errors |= 1L ≪ 22;      /* hexadecimal number not working */
  gb_newline( );      /* now we should be scanning a blank line */
  if (gb_char( ) ≠ '\n') io_errors |= 1L ≪ 23;      /* newline not inserted at end */
  if (gb_char( ) ≠ '\n') io_errors |= 1L ≪ 24;      /* newline not implied after end */
  if (gb_number(60) ≠ 0) io_errors |= 1L ≪ 25;      /* number should stop at null character */
  { char temp[100];
    if (gb_string(temp, '\n') ≠ temp + 1) io_errors |= 1L ≪ 26;
        /* string should be null after end of line */
    gb_newline( );
    if (gb_string(temp, ':') ≠ temp + 5 ∨ strcmp(temp, "Oops")) io_errors |= 1L ≪ 27;
        /* string not read properly */
  }
  if (io_errors) exit_test("Sorry, it failed. Look at the error code for clues");
  if (gb_digit(10) ≠ −1) exit_test("Digit error not detected");
  if (gb_char( ) ≠ ':') io_errors |= 1L ≪ 28;      /* lost synch after gb_string and gb_digit */
  if (gb_eof( )) io_errors |= 1L ≪ 29;      /* premature end-of-file indication */
  gb_newline( );
  if (¬gb_eof( )) io_errors |= 1L ≪ 30;      /* postmature end-of-file indication */
```

This code is used in section 2.

28. Opening a file. The call *gb_raw_open*(`"foo"`) will open file `"foo"` and initialize the checksumming process. If the file cannot be opened, *io_errors* will be set to *cant_open_file*, otherwise *io_errors* will be initialized to zero.

The call *gb_open*(`"foo"`) is a stronger version of *gb_raw_open*, which is used for standard GraphBase data files like `"words.dat"` to make doubly sure that they have not been corrupted. It returns the current value of *io_errors*, which will be nonzero if any problems were detected at the beginning of the file.

```
⟨ Test the gb_open routine; exit if there's trouble 28 ⟩ ≡
  if (gb_open("test.dat") ≠ 0) exit_test("Can't␣open␣test.dat");
```

This code is used in section 2.

29. `#define gb_raw_open gb_r_open` /* abbreviation for Procrustean external linkage */

```
⟨ gb_io.h 6 ⟩ +≡
#define gb_raw_open gb_r_open
  extern void gb_raw_open( );      /* open a file for GraphBase input */
  extern long gb_open( );      /* open a GraphBase data file; return 0 if OK */
```

30.

```
⟨ External functions 12 ⟩ +≡
  void gb_raw_open(f)
       char *f;
  {
    ⟨ Make sure that icode has been initialized 14 ⟩;
    ⟨ Try to open f 31 ⟩;
    if (cur_file) {
      io_errors = 0;
      more_data = 1;
      line_no = magic = 0;
      tot_lines = #7fffffff;      /* allow "infinitely many" lines */
      fill_buf( );
    } else io_errors = cant_open_file;
  }
```

31. Here's a possibly system-dependent part of the code: We try first to open the data file by using the file name itself as the path name; failing that, we try to prefix the file name with the name of the standard directory for GraphBase data, if the program has been compiled with `DATA_DIRECTORY` defined.

```
⟨ Try to open f 31 ⟩ ≡
  cur_file = fopen(f, "r");
#ifdef DATA_DIRECTORY
  if (¬cur_file ∧ (strlen(DATA_DIRECTORY) + strlen(f) < STR_BUF_LENGTH)) {
    sprintf(str_buf, "%s%s", DATA_DIRECTORY, f);
    cur_file = fopen(str_buf, "r");
  }
#endif DATA_DIRECTORY
```

This code is used in section 30.

32. ⟨ External functions 12 ⟩ +≡

```
long gb_open(f)
    char *f;
{
  strncpy(file_name, f, 19);      /* save the name for use by gb_close */
  gb_raw_open(f);
  if (cur_file) {
    ⟨ Check the first line; return if unsuccessful 34 ⟩;
    ⟨ Check the second line; return if unsuccessful 35 ⟩;
    ⟨ Check the third line; return if unsuccessful 36 ⟩;
    ⟨ Check the fourth line; return if unsuccessful 37 ⟩;
    gb_newline( );      /* the first line of real data is now in the buffer */
  }
  return io_errors;
}
```

33. ⟨ Private declarations 8 ⟩ +≡

```
static char file_name[20];      /* name of the data file, without a prefix */
```

34. The first four lines of a typical data file should look something like this:

```
* File "words.dat" from the Stanford GraphBase (C) 1993 Stanford University
* A database of English 5-letter words
* This file may be freely copied but please do not change it in any way!
* (Checksum parameters 5757,526296596)
```

We actually verify only that the first four lines of a data file named `"foo"` begin respectively with the characters

```
* File "foo"
*
*
* (Checksum parameters l,m)
```

where l and m are decimal numbers. The values of l and m are stored away as *tot_lines* and *final_magic*, to be matched at the end of the file.

⟨ Check the first line; return if unsuccessful 34 ⟩ ≡

```
sprintf(str_buf, "* File \"%s\"", f);
if (strncmp(buffer, str_buf, strlen(str_buf))) return (io_errors |= bad_first_line);
```

This code is used in section 32.

35. ⟨ Check the second line; return if unsuccessful 35 ⟩ ≡

```
fill_buf( );
if (*buffer ≠ '*') return (io_errors |= bad_second_line);
```

This code is used in section 32.

36. ⟨ Check the third line; return if unsuccessful 36 ⟩ ≡

```
fill_buf( );
if (*buffer ≠ '*') return (io_errors |= bad_third_line);
```

This code is used in section 32.

37. ⟨ Check the fourth line; return if unsuccessful 37 ⟩ ≡

```
fill_buf( );
if (strncmp(buffer, "*␣(Checksum␣parameters␣", 23)) return (io_errors |= bad_fourth_line);
cur_pos += 23;
tot_lines = gb_number(10);
if (gb_char( ) ≠ ',') return (io_errors |= bad_fourth_line);
final_magic = gb_number(10);
if (gb_char( ) ≠ ')') return (io_errors |= bad_fourth_line);
```

This code is used in section 32.

38. Closing a file. After all data has been input, or should have been input, we check that the file was open and that it had the correct number of lines, the correct magic number, and a correct final line. The subroutine *gb_close*, like *gb_open*, returns the value of *io_errors*, which will be nonzero if at least one problem was noticed.

```
⟨ Test the gb_close routine; exit if there's trouble 38 ⟩ ≡
  if (gb_close( ) ≠ 0) exit_test("Bad checksum, or difficulty closing the file");
```

This code is used in section 2.

39.
```
⟨ External functions 12 ⟩ +≡
  long gb_close( )
  {
    if (¬cur_file) return (io_errors |= no_file_open);
    fill_buf( );
    sprintf(str_buf, "* End of file \"%s\"", file_name);
    if (strncmp(buffer, str_buf, strlen(str_buf))) io_errors |= bad_last_line;
    more_data = buffer[0] = 0;      /* now the GB_IO routines are effectively shut down */
      /* we have cur_pos = buffer */
    if (fclose(cur_file) ≠ 0) return (io_errors |= cant_close_file);
    cur_file = Λ;
    if (line_no ≠ tot_lines + 1) return (io_errors |= wrong_number_of_lines);
    if (magic ≠ final_magic) return (io_errors |= wrong_checksum);
    return io_errors;
  }
```

40. There is also a less paranoid routine, *gb_raw_close*, that closes user-generated files. It simply closes the current file, if any, and returns the value of the *magic* checksum.

Example: The *restore_graph* subroutine in GB_SAVE uses *gb_raw_open* and *gb_raw_close* to provide system-independent input that is almost as foolproof as the reading of standard GraphBase data.

41.
```
#define gb_raw_close  gb_r_close      /* for Procrustean external linkage */
⟨ gb_io.h 6 ⟩ +≡
#define gb_raw_close  gb_r_close
  extern long gb_close( );      /* close a GraphBase data file; return 0 if OK */
  extern long gb_raw_close( );      /* close file and return the checksum */
```

42.
```
⟨ External functions 12 ⟩ +≡
  long gb_raw_close( )
  {
    if (cur_file) {
      fclose(cur_file);
      more_data = buffer[0] = 0;
      cur_pos = buffer;
      cur_file = Λ;
    }
    return magic;
  }
```

43. Index. Here is a list that shows where the identifiers of this program are defined and used.

a: 17, 24.
bad_first_line: 3, 34.
bad_fourth_line: 3, 37.
bad_last_line: 3, 39.
bad_second_line: 3, 35.
bad_third_line: 3, 36.
buffer: 8, 9, 16, 18, 22, 34, 35, 36, 37, 39, 42.
c: 12, 26.
cant_close_file: 3, 39.
cant_open_file: 3, 28, 30.
checksum_prime: 10, 17.
cur_file: 8, 9, 30, 31, 32, 39, 42.
cur_pos: 8, 9, 21, 22, 23, 24, 25, 26, 37, 39, 42.
d: 12, 24.
DATA_DIRECTORY: 31.
exit_test: 2, 27, 28, 38.
f: 30, 32.
fclose: 3, 39, 42.
fgets: 3, 9.
file_ended_prematurely: 3, 9.
file_name: 32, 33, 39.
fill_buf: 9, 18, 30, 35, 36, 37, 39.
final_magic: 10, 34, 37, 39.
fopen: 3, 31.
fprintf: 2.
gb_backup: 21, 22, 27.
gb_char: 21, 22, 27, 37.
gb_close: 32, 38, 39, 41.
gb_digit: 23, 24, 27.
gb_eof: 19, 20, 27.
gb_newline: 16, 17, 18, 21, 27, 32.
gb_number: 23, 24, 27, 37.
gb_open: 28, 29, 32, 38.
gb_r_close: 41.
gb_r_open: 29.
gb_raw_close: 40, 41, 42.
gb_raw_open: 28, 29, 30, 32, 40.
gb_string: 25, 26, 27.
icode: 10, 11, 12, 14, 15, 17, 23, 24.
icode_setup: 14, 15.
imap: 11, 12, 15.
imap_chr: 12, 13.
imap_ord: 12, 13.
io_errors: 2, 3, 5, 6, 9, 27, 28, 30, 32, 34, 35, 36, 37, 38, 39.
k: 15.
line_no: 10, 18, 30, 39.
magic: 10, 17, 18, 30, 39, 40, 42.
main: 2.
missing_newline: 3, 9.
more_data: 9, 10, 18, 19, 20, 30, 39, 42.
new_checksum: 16, 17, 18.
no_file_open: 3, 39.
old_checksum: 17.
p: 9, 15, 17, 26.
printf: 2.
restore_graph: 40.
s: 17.
sprintf: 31, 34, 39.
stderr: 2.
str_buf: 25, 26, 31, 34, 39.
STR_BUF_LENGTH: 25, 26, 31.
strcmp: 27.
strlen: 12, 31, 34, 39.
strncmp: 34, 37, 39.
strncpy: 32.
system dependencies: 31.
SYSV: 7.
temp: 27.
tot_lines: 10, 18, 30, 34, 37, 39.
unexpected_char: 11, 12, 13, 15.
wrong_checksum: 3, 39.
wrong_number_of_lines: 3, 39.

⟨ Check the first line; return if unsuccessful 34 ⟩ Used in section 32.
⟨ Check the fourth line; return if unsuccessful 37 ⟩ Used in section 32.
⟨ Check the second line; return if unsuccessful 35 ⟩ Used in section 32.
⟨ Check the third line; return if unsuccessful 36 ⟩ Used in section 32.
⟨ External declarations 5 ⟩ Used in section 4.
⟨ External functions 12, 17, 18, 20, 22, 24, 26, 30, 32, 39, 42 ⟩ Used in section 4.
⟨ Header files to include 7 ⟩ Used in sections 4 and 6.
⟨ Internal functions 9, 15 ⟩ Used in section 4.
⟨ Make sure that *icode* has been initialized 14 ⟩ Used in sections 12 and 30.
⟨ Private declarations 8, 10, 11, 33 ⟩ Used in section 4.
⟨ Test the sample data lines; exit if there's trouble 27 ⟩ Used in section 2.
⟨ Test the *gb_close* routine; exit if there's trouble 38 ⟩ Used in section 2.
⟨ Test the *gb_open* routine; exit if there's trouble 28 ⟩ Used in section 2.
⟨ Try to open *f* 31 ⟩ Used in section 30.
⟨ `gb_io.h` 6, 13, 16, 19, 21, 23, 25, 29, 41 ⟩
⟨ `test_io.c` 2 ⟩

January 12, 1994 at 23:13

GB_IO

	Section	Page
Introduction	1	1
Inputting a line	8	3
Checksums	10	4
Parsing a line	21	7
Opening a file	28	10
Closing a file	38	13
Index	43	14

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.