barcode 0.98

A library for drawing bar codes
March 2002

by **Alessandro Rubini** (rubini@gnu.org)

Barcode tools

This file documents version 0.98 of the barcode library and sample programs (March 2002).

1 Overview

The *barcode* package is mainly a C library for creating bar-code output files. It also includes a command line front-end and (in a foreseeable future) a graphic frontend.

The package is designed as a library because we think the main use for barcode-generation tools is inside more featured applications. The library addresses bar code printing as two distinct problems: creation of bar information and actual conversion to an output format. To this aim we use an intermediate representation for bar codes, which is currently documented in the 'ps.c' source file (not in this document).

Note that the library and the accompanying material is released according to the GPL license, not the LGPL one. A copy of the GPL is included in the distribution tarball.

2 The Underlying Data Structure

Every barcode-related function acts on a data structure defined in the 'barcode.h' header, which must be included by any C source file that uses the library. The header is installed by `make install`.

The definition of the data structure is included here for reference:

```
struct Barcode_Item {
    int flags;         /* type of encoding and other flags */
    char *ascii;       /* malloced */
    char *partial;     /* malloced too */
    char *textinfo;    /* information about text placement */
    char *encoding;    /* code name, filled by encoding engine */
    int width, height; /* output units */
    int xoff, yoff;    /* output units */
    int margin;        /* output units */
    double scalef;     /* requested scaling for barcode */
    int error;         /* an errno-like value, in case of failure */
};
```

The exact meaning of each field and the various flags implemented are described in the following sections.

Even though you won't usually need to act on the contents of this structure, some of the functions in the library receive arguments that are directly related to one or more of these fields.

2.1 The Fields

`int flags;`
: The flags are, as you may suspect, meant to specify the exact behaviour of the library. They are often passed as an argument to *barcode* functions and are discussed in the next section.

`char *ascii;`
`char *partial;`
`char *textinfo;`
`char *encoding;`
: These fields are internally managed by the library, and you are not expected to touch them if you use the provided API. All of them are allocated with *malloc*.

`int width;`
`int height;`

They specify the width and height of the *active* barcode region (i.e., excluding the white margin), in the units used to create output data (for postscript they are points, 1/72th of an inch, 0.352 mm). The fields can be either assigned to in the structure or via *Barcode_Position()*, at your choice. If either value or both are left to their default value of zero, the output engine will assign default values according to the specified scaling factor. If the specified width is bigger than needed (according to the scaling factor), the output barcode will be centered in its requested region. If either the width of the height are too small for the specified scale factor, the output bar code will expand symmetrically around the requested region.

`int xoff;`
`int yoff;`

The fields specify offset from the coordinate origin of the output engine (for postscript, position 0,0 is the lower left corner of the page). The fields can be either assigned to in the structure or via *Barcode_Position()*, at your choice. The offset specifies where the white margin begins, not where the first bar will be printed. To print real ink to the specified position you should set *margin* to 0.

`int margin;`

The white margin that will be left around the printed area of the bar code. The same margin is applied to all sides of the printed area. The default value for the margin is defined in '`barcode.h`' as `BARCODE_DEFAULT_MARGIN` (10).

`double scalef;`

The enlarge or shrink value for the bar code over its default dimension. The *width* and *scalef* fields interact deeply in the creation of the output, and a complete description of the issues appears later in this section.

`int error;`

The field is used when a *barcode* function fails to host an `errno`-like integer value.

Use of the *width* and *scalef* fields.

A width unit is the width of the thinnest bar and/or space in the chosen code; it defaults to 1 point if the output is postscript or encapsulated postscript.

Either or both the code width and the scale factor can be left unspecified (i.e., zero). The library deals with defaults in the following way:

Both unspecified

If both the width and the scale factor are unspecified, the scale factor will default to 1.0 and the width is calculated according to the actual width of the bar code being printed.

Width unspecified

If the width is not specified, it is calculated according to the values of *scalef*.

Scale factor unspecified

If the scale factor is not specified, it will be chosen so that the generated bar code exactly fits the specified width.

Both specified

The code will be printed inside the specified region according to the specified scale factor. It will be aligned to the left. If, however, the chosen width is too small for the specific bar code and scaling factor, then the code will extend symmetrically to the left and to the right of the chosen region.

2.2 The Intermediate Representation

The encoding functions print their output into the `partial` and `texinfo` fields of the barcode data structure. Those fields, together with position information, are then used to generate actual output. This is an informal description of the intermediate format.

The first char in `partial` tells how much extra space to add to the left of the bars. For EAN-13, it is used to leave space to print the first digit, other codes may have '0' for no-extra-space-needed.

The next characters are alternating bars and spaces, as multiples of the base dimension which is 1 unless the code is rescaled. Rescaling is calculated as the ratio from the requested width and the calculated width. Digits represent bar/space dimensions. Lower-case letters represent those bars that should extend lower than the others: 'a' is equivalent to '1', 'b' is '2' and so on up to 'i' which is equivalent to '9'. Other letters will be used for encoding-specific meanings, as soon as I implement them.

The `textinfo` string is made up of fields `%lf:%lf:%c` separated by blank space. The first integer is the x position of the character, the second is the font size (before rescaling) and the char item is the character to be printed.

Both the `partial` and `textinfo` strings may include "`-`" or "`+`" as special characters (in `textinfo` the char should be a stand-alone word). They state where the text should be printed: below the bars ("`-`", default) or above the bars. This is used, for example, to print the add-5 and add-2 codes to the right of UPC or EAN codes (the add-5 extension is mostly used in ISBN codes).

3 The Flags

The following flags are supported by version 0.98 of the library:

`BARCODE_ENCODING_MASK`
: The mask is used to extract the encoding-type identifier from the *flags* field.

`BARCODE_EAN`
`BARCODE_UPC`
`BARCODE_ISBN`
`BARCODE_128B`
`BARCODE_128C`
`BARCODE_128`
`BARCODE_128RAW`
`BARCODE_39`
`BARCODE_I25`
`BARCODE_CBR`
`BARCODE_MSI`
`BARCODE_PLS`
`BARCODE_93`
: The currently supported encoding types: EAN (13 digits, 8 digits, 13 + 2 add-on and 13 + 5 add-on), UPC (UPC-A, UPC-E, UPC-A with 2 or 5 digit add-on), ISBN (with or without the 5-digit add-on), CODE128-B (the whole set of printable ASCII characters), CODE128-C (two digits encoded by each barcode symbol), CODE128 (all ASCII values), a "raw-input" pseudo-code that generates CODE128 output, CODE39 (alphanumeric), "interleaved 2 of 5" (numeric), Codabar (numeric plus a few symbols), MSI (numeric) and Plessey (hex digits). See Chapter 6 [Supported Encodings], page 7.

`BARCODE_ANY`
: This special encoding type (represented by a value of zero, so it will be the default) tells the encoding procedure to look for the first encoding type that can deal with a textual string. Therefore, a 11-digit code will be printed as UPC (as well as 6-digit, 11+2 and 11+5), a 12-digit (or 7-digit, or 12+2 or 12+5) as EAN13, an ISBN code (with or without hyphens, with or without add-5) will be encoded in its EAN13 representation, an even number of digits is encoded using CODE128C and a generic string is encoded using CODE128B. Since code-39 offers a much larger representation for the same text string, code128-b is preferred over code39 for alphanumeric strings.

`BARCODE_NO_ASCII`
: Instructs the engine not to print the ascii string on output. By default the bar code is accompanied with an ascii version of the text it encodes.

`BARCODE_NO_CHECKSUM`
: Instructs the engine not to add the checksum character to the output. Not all the encoding types can drop the checksum; those where the checksum is mandatory (like EAN and UPC) just ignore the flag.

`BARCODE_OUTPUT_MASK`
: The mask is used to extract the output-type identifier from the *flags* field.

`BARCODE_OUT_PS`
`BARCODE_OUT_EPS`
`BARCODE_OUT_PCL`
`BARCODE_OUT_PCL_III`
: The currently supported encoding types: full-page postscript and encapsulated postscript; PCL (print command language, for HP printers) and PCL-III (same as PCL, but uses a font not available on older printers).

`BARCODE_OUT_NOHEADERS`
: The flag instructs the printing engine not to print the header and footer part of the file. This makes sense for the postscript engine but might not make sense for other engines; such other engines will silently ignore the flag just like the PCL back-end does.

4 Functions Exported by the Library

The functions included in the barcode library are declared in the header file `barcode.h`. They perform the following tasks:

`struct Barcode_Item *Barcode_Create(char *text);`
: The function creates a new barcode object to deal with a specified text string. It returns NULL in case of failure and a pointer to a barcode data structure in case of success.

`int Barcode_Delete(struct Barcode_Item *bc);`
: Destroy a barcode object. Always returns 0 (success)

`int Barcode_Encode(struct Barcode_Item *bc, int flags);`
: Encode the text included in the *bc* object. Valid flags are the encoding type (other flags are ignored) and BARCODE_NO_CHECKSUM (other flags are silently ignored); if the flag argument is zero, `bc->flags` will apply. The function returns 0 on success and -1 in case of error. After successful termination the data structure will host the description of the bar code and its textual representation, after a failure the `error` field will include the reason of the failure.

`int Barcode_Print(struct Barcode_Item *bc, FILE *f, int flags);`
: Print the bar code described by `bc` to the specified file. Valid flags are the output type, `BARCODE_NO_ASCII` and `BARCODE_OUT_NOHEADERS`, other flags are ignored. If any of these flags is zero, it will be inherited from `bc->flags` which therefore takes precedence. The function returns 0 on success and -1 in case of error (with `bc->error` set accordingly). In case of success, the bar code is printed to the specified file, which won't be closed after use.

`int Barcode_Position(struct Barcode_Item *bc, int wid, int hei, int xoff, int yoff, double scalef);`
: The function is a shortcut to assign values to the data structure.

`int Barcode_Encode_and_Print(char *text, FILE *f, int wid, int hei, int xoff, int yoff, int flags);`
: The function deals with the whole life of the barcode object by calling the other functions; it uses all the specified flags.

`int Barcode_Version(char *versionname);`
: Returns the current version as an integer number of the form major * 10000 + minor * 100 + release. Therefore, version 1.03.5 will be returned as 10305 and version 0.53 as 5300. If the argument is non-null, it will be used to return the version number as a string. Note that the same information is available from two preprocessor macros: `BARCODE_VERSION` (the string) and `BARCODE_VERSION_INT` (the integer number).

5 The *barcode* frontend program

The **barcode** program is a front-end to access some features of the library from the command line. It is able to read user supplied strings from the command line or a data file (standard input by default) and encode all of them.

5.1 The Command Line

barcode accepts the following options:

`--help or -h`
: Print a usage summary and exit.

`-i filename`
: Identify a file where strings to be encoded are read from. If missing (and if `-b` is not used) it defaults to standard input. Each data line of the input file will be used to create one barcode output.

`-o filename`
: Output file. It defaults to standard output.

`-b string`
: Specify a single "barcode" string to be encoded. The option can be used multiple times in order to encode multiple strings (this will result in multi-page postscript output or a table of barcodes if `-t` is specified). The strings must match the encoding chosen; if it doesn't match the program will print a warning to `stderr` and generate "blank" output (although not zero-length). Please note that a string including spaces or other special characters must be properly quoted.

`-e encoding`
: **encoding** is the name of the chosen encoding format being used. It defaults to the value of the environment variable `BARCODE_ENCODING` or to auto detection if the environment is also unset.

`-g geometry`
: The geometry argument is of the form "[<*width*> `x` <*height*>] [`+` <*xmargin*> `+` <*ymargin*>]" (with no intervening spaces). Unspecified margin values will result in no margin; unspecified size results in default size. The specified values represent print points by default, and can be inches, millimeters or other units according to the `-u` option or the `BARCODE_UNIT` environment variable. The argument is used to place the printout code on the page. Note that an additional white margin of 10 points is added to the printout. If the option is unspecified, `BARCODE_GEOMETRY` is looked up in the environment, if missing a default size and no margin (but the default 10 points) are used.

`-t table-geometry`
: Used to print several barcodes to a single page, this option is meant to be used to print stickers. The argument is of the form "<*columns*> `x` <*lines*> [`+` <*leftmargin*> `+` <*bottommargin*> [`-` <*rightmargin*> [`-` <*topmargin*>]]]" (with no intervening spaces); if missing, the top and right margin will default to be the same as the bottom and left margin. The margins are specified in print points or in the chosen unit (see `-u` below). If the option is not specified, `BARCODE_TABLE` is looked up in the environment, otherwise no table is printed and each barcode will get its own page. The size (but not the position) of a barcode item within a table can also be selected using `-g` (see "geometry" above), without struggling with external and internal margins. I still think management of geometries in a table is suboptimal, but I can't make it better without introducing incompatibilities.

`-m margin(s)`
: Specifies an internal margin for each sticker in the table. The argument is of the form "<*xmargin*>`,`<*ymargin*>" and the margin is applied symmetrically to the sticker. If unspecified, the environment variable `BARCODE_MARGIN` is used or a default internal margin of 10 points is used.

`-n`
: "Numeric" output: don't print the ASCII form of the code, only the bars.

`-c`
: No checksum character (for encodings that allow it, like code 39, other codes, like UPC or EAN, ignore this option).

`-E`
: Encapsulated postscript (default is normal postscript). When the output is generated as EPS only one barcode is encoded.

`-P`
: PCL output. Please note that the Y direction goes from top to bottom for PCL, and the origin for an image is the top-left corner instead of the bottom-left

`-p pagesize`
: Specify a non-default page size. The page size can be specified in millimeters, inches or plain numbers (for example: `"210x297mm"`, `"8.5x11in"`, `"595x842"`). A page specification as numbers will be interpreted according to the current unit specification (see `-u` below). If libpaper is available, you can also specify the page size with its name, like `"A3"` or `"letter"` (libpaper is a standard component of Debian GNU/Linux, but may be missing elsewhere). The default page size is your system-wide default if libpaper is there, A4 otherwise.

`-u unit`
: Choose the unit used in size specifications. Accepted values are "mm", "cm", "in" and "pt". By default, the program will check `BARCODE_UNIT` in the environment, and assume points otherwise (this behaviour is compatible with 0.92 and previous versions. If `-u` appears more than once, each instance will modified the behaviour for the arguments at its right, as the command line is processes left to right. The program internally works with points, and any size is approximated to the nearest multiple of one point. The `-u` option affect `-g` (geometry), `-t` (table) and `-p` (page size).

6 Supported Encodings

The program encodes text strings passed either on the command line (with -b) or retrieved from standard input. The text representation is interpreted according to the following rules. When auto-detection of the encoding is enabled (i.e, no explicit encoding type is specified), the encoding types are scanned to find one that can digest the text string. The following list of supported types is sorted in the same order the library uses when auto-detecting a suitable encoding for a string.

EAN
: The EAN frontend is similar to UPC; it accepts strings of digits, 12 or 7 characters long. Strings of 13 or 8 characters are accepted if the provided checksum digit is correct. I expect most users to feed input without a checksum, though. The add-2 and add-5 extension are accepted for both the EAN-13 and the EAN-8 encodings. The following are example of valid input strings: "`123456789012`" (EAN-13), "`1234567890128`" (EAN-13 wih checksum), "`1234567`" (EAN-8), "`12345670 12345`" (EAN-8 with checksum and add-5), "`123456789012 12`" (EAN-13 with add-2), "`123456789012 12345`" (EAN-13 with add-5).

UPC
: The UPC frontend accepts only strings made up of digits (and, if a supplemental encoding is used, a blank to separate it). It accepts strings of 11 or 12 digits (UPC-A) and 6 or 7 or 8 digits (UPC-E).

  The 12th digit of UPC-A is the checksum and is added by the library if not specified in the input; if it is specified, it must be the right checksum or the code is rejected as invalid. For UPC-E, 6 digit are considered to be the middle part of the code, a leading 0 is assumed and the checksum is added; 7 digits are either considered the initial part (leading digit 0 or 1, checksum missing) or the final part (checksum specified, leading 0 assumed); 8 digits are considered to be the complete code, with leading 0 or 1 and checksum. For both UPC-A and UPC-E, a trailing string of 2 digits or 5 digits is accepted as well. Therefore, the following are examples of valid strings that can be encoded as UPC: "`01234567890`" (UPC-A) "`012345678905`" (UPC-A with checksum), "`012345`" (UPC-E), "`01234567890 12`" (UPC-A, add-2) and "`01234567890 12345`" (UPC-A, add-5), "`0123456 12`" (UPC-E, add-2). Please note that when setting `BARCODE_ANY` to auto-detect the encoding to be used, 12-digit strings and 7-digit strings will always be identified as EAN. This because I expect most user to provide input without a checksum. If you need to specify UPC-with-checksum as input you must explicitly set `BARCODE_UPC` as a flag or use `-e upc` on the command line.

ISBN
: ISBN numbers are encoded as EAN-13 symbols, with an optional add-5 trailer. The ISBN frontend of the library accepts real ISBN numbers and deals with any hyphen and, if present, the ISBN checksum character before encoding data. Valid representations for ISBN strings are for example: "`1-56592-292-1`", "`3-89721-122-X`" and "`3-89721-122-X 06900`".

code 128-B
: This encoding can represent all of the printing ASCII characters, from the space (32) to DEL (127). The checksum digit is mandatory in this encoding.

code 128-C
: The "C" variation of Code-128 uses Code-128 symbols to represent two digits at a time (Code-128 is made up of 104 symbols whose interpretation is controlled by the start symbol being used). Code 128-C is thus the most compact way to represent any even number of digits. The encoder refuses to deal with an odd number of digits because the caller is expected to provide proper padding to an even number of digits. (Since Code-128 includes control symbols to switch charset, it is theoretically possible to represent the odd digit as a Code 128-A or 128-B symbol, but this tool doesn't currently implement this option).

code 128 raw
: Code-128 output represented symbol-by-symbol in the input string. To override part of the problems outlined below in specifying code128 symbols, this pseudo-encoding allows the used to specify a list of code128 symbols separated by spaces. Each symbol is represented by a number in the range 0-105. The list should include the leading character.The checksum and the stop character are automatically added by the library. Most likely this pseudo-encoding will be used with `BARCODE_NO_ASCII` and some external program to supply the printed text.

code 39
: The code-39 standard can encode uppercase letters, digits, the blank space, plus, minus, dot, star, dollar, slash, percent. Any string that is only composed of such characters is accepted by the code-39 encoder. To avoid loosing information, the encoder refuses to encode mixed-case strings (a lowercase string is nonetheless accepted as a shortcut, but is encoded as uppercase).

interleaved 2 of 5
: This encoding can only represent an even number of digits (odd digits are represented by bars, and even digits by the interleaving spaces). The name stresses the fact that two of the five items (bars or spaces) allocated to each symbol are wide, while the rest are narrow. The checksum digit is optional (can be disabled via `BARCODE_NO_CHECKSUM`). Since the number of digits, including the checksum, must be even, a leading zero is inserted in the string being encoded if needed (this is specifically stated in the specs I have access to).

code 128
: Automatic selection between alphabet A, B and C of the Code-128 standard. This encoding can represent all ASCII symbols, from 0 (NUL) to 127 (DEL), as well as four special symbols, named F1, F2, F3, F4. The set of symbols available in this encoding is not easily represented as input to the *barcode* library, so the following convention is used. In the input string, which is a C-language null-terminated string, the NUL char is represented by the value 128 (0x80, 0200) and the F1-F4 characters are represented by the values 193-196 (0xc1-0xc4, 0301-0304). The values have been chosen to ease their representation as escape sequences.

  Since the shell doesn't seem to interpret escape sequences on the command line, the "-b" option cannot be easily used to designate the strings to be encoded. As a workaround you can resort to the command `echo`, either within back-ticks or used separately to create a file that is then fed to the standard-input of *barcode* – assuming your `echo` command processes escape sequences. The newline character is especially though to encode (but not impossible unless you use a `csh` variant.

  These problems only apply to the command-line tool; the use of library functions doesn't give any problem. In needed, you can use the "*code 128 raw*" pseudo-encoding to represent code128 symbols by their numerical value. This encoding is used late in the auto-selection mechanism because (almost) any input string can be represented using code128.

Codabar
: Codabar can encode the ten digits and a few special symbols (minus, plus, dollar, colon, bar, dot). The characters "`A`", "`B`", "`C`" and "`D`" are used to represent four different start/stop characters. The input string to the barcode library can include the start and stop characters or not include them (in which case "`A`" is used as start and "`B`" as stop). Start and stop characters in the input string can be either all lowercase or all uppercase and are always printed as uppercase.

Plessey
: Plessey barcodes can encode all the hexadecimal digits. Alphabetic digits in the input string must either be all lowercase or all uppercase. The output text is always uppercase.

MSI MSI can only encode the decimal digits. While the standard specifies either one or two check digits, the current implementation in this library only generates one check digit.

code 93 The code-93 standard can natively encode 48 different characters, including uppercase letters, digits, the blank space, plus, minus, dot, star, dollar, slash, percent, as well as five special characters: a start/stop delimiter and four "shift characters" used for extended encoding. Using this "extended encoding" method, any standard 7-bit ASCII character can be encoded, but it takes up two symbol lengths in barcode if the character is not natively supported (one of the 48). The encoder here fully implements the code 93 encoding standard. Any characters natively supported (A-Z, 0-9, ".+-/$&%") will be encoded as such - for any other characters (such as lower case letters, brackets, parentheses, etc.), the encoder will revert to extended encoding. As a note, the option to exclude the checksum will eliminate the two modulo-47 checksums (called C and K) from the barcode, but this probably will make it unreadable by 99% of all scanning systems. These checksums are specified to be used at the firmware level, and their absence will be interpreted as an invalid barcode.

7 PCL Output

While the default output is Postscript (possibly EPS), and Postscript can be post-processed to almost anything, it is sometimes desirable to create output directly usable by the specific printer at hand. PCL is currently supported as an output format for this reason. Please note that the Y coordinate for PCL goes from top to bottom, while for Postscript it goes from bottom to top. Consistently, while in Postscript you specify the bottom-left corner as origin, for PCL you specify the top-left corner.

Barcode output for PCL Printers (HP LaserJet and compatibles), was developed using PCL5 Reference manuals from HP. that really refers to these printers:

- LaserJet III, III P, III D, III Si,
- LaserJet 4 family
- LaserJet 5 family
- LaserJet 6 family
- Color LaserJet
- DeskJet 1200 and 1600.

However, barcode printing uses a very small subset of PCL, probably also LaserJet II should print it without problem, but the resulting text may be horrible.

The only real difference from one printer to another really depends on which font are available in the printer, used in printing the label associated to the bars (if requested).

Earlier LaserJet supports only bitmaps fonts, so these are not "scalable". (Ljet II ?), Also these fonts, when available, have a specified direction, and not all of them are available in both Portrait and Landscape mode.

From LaserJet 4 series, (except 4L/5L that are entry-level printers), Arial scalable font should be available, so it's the "default font" used by this program.

LaserJet III series printers (and 4L, 5L), don't feature "Arial" as a resident font, so you should use `BARCODE_OUT_PCL_III` instead of `BARCODE_OUT_PCL.`, and font the font used will be "Univers" instead of "Arial".

Results on compatible printers, may depend on consistency of PCL5 compatibility, in doubt, try BARCODE_OUT_PCL_III

PJL commands are not used here, as it's not very compatible.

Tested Printers:

- Hp LaserJet 4050
- Hp LaserJet 2100
- Epson N-1200 emul PCL
- Toshiba DP2570 (copier) + PCL option
- Epson EPL-7100 emul. HP LaserJet II: bars print fine but text is bad.

8 Bugs and Pending Issues.

The current management of borders/margins is far from optimal. The "default" margin applied by the library interferes with the external representation, but I feel it is mandatory to avoid creating barcode output with no surrounding white space (the problem is especially relevant for EPS output).

EAN-128 is not (yet) supported. I plan to implement it pretty soon and then bless the package as version 1.0.

Table of Contents