The hibLib engines - draft

February 19, 2006

Contents

Introduction

About this document

This document will describe the API of the hibLib library.

Chapter 1

Formatted text drawing engine

1.1 The main processus

The HibLib main engine works in sequence of two distinguish process. The first work can be called the parsing, and the second the formatting of the text to show. When you open a TEXT file with HibView, you can see the two pass. The fisrt pass is desingned to read the text and to create objects independantly of the calculator. Then the formatting organize every object in "screen lines", depending of the calculator.

1.1.1 1st pass

In fact the first pass reads the initial text and tries to interpret every format and defined object as `#U`, `#2`, `&E`, etc... This first pass creates a table of "Object". This table is stored in the `h_File` structure (`h_objs`, `hobjs` and `nb_objs`). Here is the definition of an "Object":

```
/* The different objects */
typedef enum {
  HOBJECT_TIOS_LINE     = 0,
  HOBJECT_TEXT          = 1,
  HOBJECT_PIC           = 2,
  HOBJECT_PPRINT        = 4,
  HOBJECT_LINK          = 6,
  HOBJECT_SEPARAT1      = 7,
  HOBJECT_SEPARAT2      = 8,
  HOBJECT_END_TEXT      = 9
} h_ObjectType;

/* The description of an Object */
typedef struct {
  h_ObjectType type;
  union {
    h_Format frt;         //for the text
    h_FrtLine frtline;    //for the begining of tios line
    struct {
```

```
        char wrong:1,
        comp:1;
        HANDLE handle;     //for pic or pprint kept in mem
      } pic;
  } datas;
  unsigned short pos_txt;
} h_Object;
```

Each object has a position and a type, some can have some data. So, every picture, link or pretty-print expression is an object. For the text, the definition of an object is a sequence of characters with no change of format. For instance, this is a unique `h_Object`: `"this␣is␣a␣text"`, but there are three `h_Objects` here: `"this␣#Uis#U␣a␣text"` (`"this␣"` and `"is"` and `"␣a␣text"`). The consequence is that each `h_Object` which is a `HOBJECT_TEXT` has is own format.

Other special `h_Object` have been introduced: the Ti-OS line (`HOBJECT_TIOS_LINE`) and the end of text (`HOBJECT_END_TEXT`). Note that theses lines are Ti-OS lines, and not graphical lines, so it is independant of the screen size of the calculator. Note that the `HOBJECT_TIOS_LINE` specify the beginning of a line, not a end of line. So, a bookmark beeing defined for each Ti-OS line, each bookmark will be associated with this type of `h_Object`.

1.1.2 2nd pass

The text being parsed, and every formats being read, the last thing to do is to organize the text. This pass is quite simple: it tries to put every `h_Object` in "screen lines". So this pass creates a new table of `h_Scrlines`, stored in the `hTxt` structure too.

```
typedef struct {
  unsigned short pos_txt;
  unsigned char nb_space;
  unsigned char width;
  unsigned char exp_max;
  unsigned char height;
} h_ScrLine;
```

Each screen line is defined by a position in the text. To draw more quickly, some graphical information are kept in the structure: `width`, `nb_space`...

Note that a `h_ScrLine` can start in a middle of an `h_Object`. The only example is if the `h_Object` is a `HOBJECT_TEXT`. For very long text without format changing, the `pos_txt` of a `h_ScrLine` can point in the middle of the `h_Object`.

1.2 Drawing

1.2.1 The characters

This engine draw the characters with the font engine (see section 4.2). So every character have to be by default separated by one pixel. The space between two words is defined by the size of the character which has the ASCII code 20 ; in fact the real size between two words is `size(' ')+1`.

1.2.2 The positions of the texts

The main rule of drawing is that there is a main line of drawing and every element of the line is calculated relatively of the line. On this main line is diplayed the normal text. For every exposant, the line of drawing is at the half of the text size upon the main line of drawing. For the suffix, it is the same but under the main line of drawing.

Figure 1.1: The position of the graphic elements

Note that even if the size of font changes between the normal text and the exposant one (or the suffix one), the reference size to calculate the exposant line of drawing is the size of the last normal text. For exemple, the exposant line is at the half size of "text" in the middle font for `"#2text#E#1exp#E"`, `"#2text#1#Eexp#E"` or `"#2text#3#Dsuf#D#1#Eexp#E"`.

1.2.3 The drawing modes

This library support different mode of drawing, here are their specifications.

Normal

Underline (dotted or not)

Vector

Conjug

Inversed

Italic

Bold

Conclusion about sizes

The width of the printed character is `font.width[charact]`

- `+1` if there is un character before
- `+1` if it is a vector and the folowing character is not a vector:
- `+1` if it is bold

- `+1` if it is shadow
- `+(font.height/3)` if is italic and if the preceding one is not italique, or there is no preceding character

The height upon the line of text is `font.height`

- `+yoffset+1` if it is an exposant
- `-yoffset` if it is a suffix
- `+4` if it is a vector
- `+2` if it is a conjug

The height under the line of text is `0`

- `-yoffset-1` if it is an exposant
- `+yoffset` if it is a suffix
- `+2` if it is underlined

1.2.4 Justified paragraph

For a justified paragraph, the size of the space will be variable depending of the line and depending of the used font. The algorithm is the folowing one :

1. compute the total size of every space in the line as it is a normal paragraph (not justified): $size\_space$
2. compute the size between the last character and the end of the line : end_size
3. compute the factor of enlargement : $factor = end\_size/size\_space$
4. every space between two words will be augmented by this previous factor : $new\_space\_size = old\_space_size * factor$

1.2.5 Word Warp

The Word Warp mode desable the possibility to finish a line in the middle of a word. The rule of "cutting" are the folowing ones :

- "cutting" is allowed just after the "space" character
- "cutting" is allowed just after the "-" character
- "cutting" is allowed just before or just after a link object
- adding a "space" charater on a line is always allowed

1.2.6 The objects

The pictures

The pretty-print expressions

The link

1.3 The HTXT format

The idea of the HTXT format is to make faster the load of a text. This will be a format of a Ti-OS variable which will be saved as any other TEXT file, but not editable. In fact, it will be an "image" of every object generated during the pass. So, in this format, there will be the table of `h_Object`, the table of `h_Bookmark`, etc... and the initial text with every character of format (`#1`, `#U`, `&E`,...) removed (they become useless).

The problem is that the second pass is calculator dependant. So the HTXT format will be generated by the first pass, and it will need the second pass to really draw the formatted text.

By the way, we can imagine two more formats, which be the HT89 and the HT92 format, which will be calculator dependant (in fact size-screen dependant). To continue in this idea, a large screen calculator (Ti92+ or V200) will be able to read a HT89 file, and a small screen calculator (Ti89) will read the HT92 with the possibility to scroll horizontally.

Yet, theses functionnalities are partially implemented, and the folowing specifications can change.

1.3.1 The HTXT/HT89/HT92 file implementations

The format of the file is not determined yet.

1.3.2 Defined position in the text

Chapter 2

Scrolling engine

2.1 The scroll

The scrolling engine is based on one buffer that represent the real text to draw. If the wanted text size is 30 pixels width, then the buffer size will be 30 pixels width. And if the wanted text size is 3000 pixels width, then the buffer size will be 3000 pixels width. The height of this buffer is by defaut equal to $LCD\_HEIGHT + 2 * heightest\_line$.

Having this buffer, the first time, the buffer will be filled entirely with the lines to draw. To see it on the screen, a copy of the seeable part of the buffer is copied to the LCD memory. Here we have a screen filled with the first seeable lines.

When the user will want to see the other parts of the text, there will be the scrolling mechanism. Here, two cases are possible : the pixel line we want to see is already in the buffer or not.

If the pixel line is already in the buffer, then the scrolling engine will scroll the screen (not the buffer !) in the proper direction, and will copy the missing pixel line from the buffer to the screen. At this point, the screen is scrolled.

If the wanted line is missing in the buffer, then the buffer need to be scrolled too. As the screen is scrolled pixel by pixel, the buffer is scrolled by entire text lines. After the buffer is scrolled in the proper way, the lines we wanted is now here, so we can return to the previous case.

For the page scrolling, the mecanism is nearly the same. The difference is that we request to the scrolling engine to scroll of more than on pixel : in fact, the page is scrolled of $screen\_height - common\_band$ pixels ($common\_band$ is the height in pixel we want to be common between two following pages).

This scrolling mecanism is done for the foor directions (top, bottom, left and right) is the same way.

The special scrolling functions which go to the last page or to the first is done by the `hl_goTo` function (this function goes to the specified line).

Chapter 3

The screen

Due to the special scrolling engine, every graphic function have to know the width of the screen buffer in byte unit. So, every graphic function has in parameter a structure that defines the screen memory :

```
typedef struct {
  char * ptr;
  short byte_width;
} h_ScreenMem;
```

For exemple, the classical LCD screen is defined by :

```
#define LCD_SCREEN (h_ScreenMem){.ptr=LCD_MEM,.byte_width=30}
```

The screen size is defined as a structure too :

```
typedef struct {
  short width;
  short height;
} h_ScreenSize;
```

Note that the width can be not equal to $8 * byte\_width$. This is the case for the LCD buffer for the Ti 89.

And the position of the screen :

```
typedef struct {
  short x;
  short y;
} h_ScreenPos;
```

The position is used to defined a frame in the buffer of the screen. For exemple, if you want to define a new screen in the LCD screen with a border of 10 pixels : its x position will equal to 10, y to 10, its width to $LCD\_WIDTH - 20$ and its height to $LCD\_HEIGHT - 20$ (its memory is still LCD_SCREEN).

And then the global definition of the screen

```
typedef struct {
  h_ScreenMem mem;
```

```
  h_ScreenSize size;
  h_ScreenPos pos;
} h_Screen;
```

With the previous example, it will be :

```
h_Screen internal_frame = {
  .mem = LCD_SCREEN,
  .size = {
    .width = LCD_WIDTH - 20,
    .height = LCD_HEIGHT - 20
  },
  .pos = {
    .x = 10,
    .y = 10
  }
};
```

Chapter 4

Font engine

In the HibLib library, a font engine is provided. This engine is dedicated to manipulate different font on the Ti. At the basis, the Ti as three sorts of font: the little, the medium and the large. Now, with this engine, you will be able to make your own font. Then you will be able to load the font of your choice and draw a string with this font. Some special format are included in the drawing routines, as bold and italic format.

Another feature of this engine is that it draws string faster than Ti-OS routines does. So, even if you don't use special font, you can use the HibLib routines to draw string fast.

4.1 The datas

The datas of the font are stored in variables of the Ti, with the extension "FONT". One of this variable will describe only one size of font. In consequence, if you want to describe the three Ti-OS fonts, three variables will be necessary.

The format of the variable is quite simple:

- 2B: the size of the variable (as for every variables)
- 2B: the version of description of the font variable format
- 1B: the maximum width of a character
- 1B: the maximum height of a character
- the table of width of character: table of 256 elements of 1B
- the table of picture of character: table of 256 elements of $(height * (width/8))$ Bytes
- a string which describe the font (for example `"TI-OS␣Font␣1"`)
- a string of the name of the author (for example `"Me"`)
- the FONT tag: `"FONT\0\xF8"`

4.2 Drawing a font

The sprite of a character will describe the entire character and not the defined space detween two characters. For exemple, the little Ti-OS font have one pixel free between two characters. In the Ti-OS, theses sprites are stored with the space : the character 'e' has a width of 4 pixels for the Ti-OS. In fact, there is 3 pixels for the 'e' and 1 pixel for the space between the 'e' and the next charater. In the HibDLL, the width of the 'e' charater will be 3 pixels. The routine which draw strings will automatically add 1 pixel between each character.

For the space between two lines of strings, this is the same idea. For the Ti-OS, the little font have a height of 6 pixels. HibDLL will consider the little font having a height of 5 pixels.

This difference has been implemented to have more precision in drawing. For exemple, drawing justified paragraph will be more clean : the last caharter of a line won't finish with a free pixel.

4.3 Load a font

Before using a font, you have to load the font. In fact, it simply search the FONT variable, gets its handle, locks it, and fills the `h_Font` structure.

```
typedef struct {
  HANDLE h;
  unsigned char filename[18];
  unsigned char * name;
  unsigned char x;
  unsigned char y;
  unsigned char * xTable;
  unsigned char * spriteTable;
} h_Font;
```

So, because it locks the handle, you have to unload a font before leaving your software: use the macro `h_unloadFont(font)` to do it.

4.4 Load every font

Another way to load font is to use the `h_loadAllFont` function. It search in the Ti every FONT variable and store the `h_Font` structures in a table. Then when you want a specific font, you just have to use `h_findFont` which return the index of the table of the font you want.

The table is described by the folowing structure:

```
typedef struct {
  h_Font * tab;
  short nb;
} h_FontTab;
```

By this way too, you have to unload every FONT by using the macro `h_unloadAllFont(fonttab)`.

Chapter 5

Menu engine

The Menu engine is an engine which provides an easy control of scrolling menus and trees. This engine is for example used for the bookmark menu. The configuration of the engine consist only in specifying the entries of the menu (and eventually their level for trees), the size of your menu and the behaviour relatively to the pressed keys. Then the engine will automatically draw the menu and wait for a choice of the user. Finnaly, you just have to handle with the choice of the user and not his "moving" in the menu.

5.1 Initialisation

To specify the behaviour of the menu, you have to fill the `h_Menu` structure:

```
typedef struct st_h_Menu h_Menu;

struct st_h_Menu {
//the datas
  short nb;
  short size_item;
  char * tab;
  h_MenuLevel * level_tab;
//the behaviour
  unsigned short key_able;
//drawing config
  h_Font * font;
  unsigned char pos_x;
  unsigned char pos_y;
  unsigned char width;
  unsigned char nb_draw;
  h_fctDrawMenu fct_draw;
//states variables
  short no_choice;
  unsigned char top;
  char move;
};
```

Desciption of the `h_Menu` structure:

`nb` the number of entries

`size_item` the size of one item: for example, if you have a table of pointer on strings, you should enter `sizeof(char *)`

`tab` the table of item: by default, it's a table of pointer of string, but we will be able to force the cast (see 5.5)

`level_tab` the table of the levels of the entries

`key_able` the keys that should be accepted while browsing the menu (see 5.2)

`font` the font which will be used for drawing the entries

`pos_x` and `pos_y` defines the top left position of the menu

`width` defines the width of the menu

`nb_draw` defines the number of entry to draw on the screen

`fct_draw` defines the drawing function used to draw the entries

`no_choice` it is the current choice of the user

`top` it is the number of the entru drawn on the top of the menu

`move` it can have the values 0 (false) or 1 (true): it describes if it is in a "move mode" or not (enabled with the `HMENU_ENABLE_MOVE` see 5.3)

5.2 Define the enabled keys

You can choose the keys you want to handle, and the key you don't want to use. Here is the definition of the keys which are supported:

```
/* code to enable the key for the menu */
#define HMENU_ENABLE_RIGHT   0x0001
#define HMENU_ENABLE_LEFT    0x0002
#define HMENU_ENABLE_Fx      0x007C //every Fx key
#define HMENU_ENABLE_F1      0x0004
#define HMENU_ENABLE_F2      0x0008
#define HMENU_ENABLE_F3      0x0010
#define HMENU_ENABLE_F4      0x0020
#define HMENU_ENABLE_F5      0x0040
#define HMENU_ENABLE_DEL     0x0080
#define HMENU_ENABLE_MOVE    0x0100
#define HMENU_ENABLE_MODE    0x0200
```

So if you just want to use the F1 and F4 key, you should write:

```
hMenu mymenu = { .nb=...
                 .key_able=HMENU_ENABLE_F1 & HMENU_ENABLE_F4,
                 ...
               };
```

5.3 The "move" mode

The engine allows you to enter in a "move" mode. This mode allows the user to move entries in the menu. For exemple, you can imagine a "Favoris" menu, and the user wants to reorder the menu at his wantings. So there is a special mode, enabled by the APPS key. The user enter in this mode by pressing the key APPS on the entry he wants to move. Then, every movement (up and down) will move the entry in the menu. He can cancel his moving by pressing ESC, and he can validate his moving by pressing the APPS key again.

To allow a such mode in your menu, you have to specify that the move mode is enabled by adding the `HMENU_ENABLE_MOVE` to the `key_able`.

5.4 The levels

The menu engine can works with levels. The level of each item is decribed by this folowing structure:

```
typedef struct {
  char level:6,
       hide:1,
       draw:1;
} h_MenuLevel;
```

As the definition of the structure shows, `level` starts from 0 (the root level) to 2$\hat{6}$-1=63 (the most little leaf). Another field `hide` describes if the item should be hide or not and the last field `draw` is internally used to know if an entry should be drawn or not.

If we want to do:

```
+ l_1
  + l_11
  + l_12
    + l_121
  + l_13
+ l_2
+ l_3
  + l_31
  + l_32
```

So the structure should be:

```
h_MenuLevel level_tab[} = {{.level=0, //l_1
                            .hide=0,
                            .draw=1},
                           {.level=1, //l_11
```

```
                                .hide=0,
                                .draw=1},
                               {.level=1, //l_12
                                .hide=0,
                                .draw=1},
                               {.level=2, //l_121
                                .hide=0,
                                .draw=1},
                               {.level=1, //l_13
                                .hide=0,
                                .draw=1},
                               {.level=0, //l_2
                                .hide=0,
                                .draw=1},
                               {.level=0, //l_3
                                .hide=0,
                                .draw=1},
                               {.level=1, //l_31
                                .hide=0,
                                .draw=1},
                               {.level=1, //l_32
                                .hide=0,
                                .draw=1}
                              };
```

Then if you want to hide every subnode of `l_1`:

```
h_MenuLevel level_tab[] = {{.level=0, //l_1
                                .hide=0,
                                .draw=1},
                               {.level=1, //l_11
                                .hide=1,
                                .draw=0},
                               {.level=1, //l_12
                                .hide=1,
                                .draw=0},
                               {.level=2, //l_121
                                .hide=1,
                                .draw=0},
                               {.level=1, //l_13
                                .hide=1,
                                .draw=0},
                               {.level=0, //l_2
                                .hide=0,
                                .draw=0},
                               {.level=0, //l_3
                                .hide=0,
                                .draw=0},
                               {.level=1, //l_31
                                .hide=0,
                                .draw=0},
```

```
                                {.level=1, //l_32
                                 .hide=0,
                                 .draw=0}
                               };
```

The engine will only draw:

```
+ l_1
+ l_2
+ l_3
  + l_31
  + l_32
```

5.5 The custum drawing function

For special application, the entries of the menu can be other object than strings. The engine can only draw item that are strings, so yo should provide a function to draw the item. A such function should have the folowing type:

```
typedef void (*h_fctDrawMenu) (short i, short x, short y, h_Menu * hmenu,
                               h_ScreenMem screen);
```

The parameters of the function are:

`i`: is the number of the item

`x` and `y`: this is the position of the entry in the screen

`h_menu`: it is the `h_Menu` structure which is used for the menu (so you can have access to the `tab`)

`screen`: this is the screen where the item would be drawn

An exemple of this type of use is the "home" menu of HibView. This menu show the variables of the Ti as a tree. In fact, the `tab` used for this menu is a table of `HSym`. Then, to draw correctly the name of the variable, and further more some additionnal informations of the selected object, the folowing drawing function is passed to the menu engine.

```
void h_drawVAT(short i, short x, short y, h_Menu * hmenu,
               h_ScreenMem screen) {
  HSym * tabhsym=(HSym *)(hmenu->tab);
  unsigned char name[20];
  unsigned char buff[20];
  const unsigned char * ext =NULL;
  char * data = NULL;
  short type;
  SYM_ENTRY * SymPtr = DerefSym(tabhsym[i]);
  SymCpy0(name,SymPtr->name);

  if (SymPtr->flags.bits.folder) {
    ext="Folder";
    strcpy(buff,name);
```

```
15    } else {
        data = HeapDeref(SymPtr->handle);
        type = h_getFileType(data);
        switch(type) {
          case UNKNOWN_FILE:
20          ext="???";
            break;
          case TEXT_FILE:
            ext="txt";
            break;
25        case HTXT_FILE:
            ext="htxt";
            break;
          case PIC_FILE:
            ext="pic";
30          break;
          case STR_FILE:
            ext="str";
            break;
        }
35      sprintf(buff,"%s.%s",name,ext);
      }

      hl_drawStr(hmenu->font,x,y,buff,FALSE,FALSE,screen);

40    if (hmenu->no_choice==i) {
        hl_fillFrame(LCD_WIDTH/2+5,15,LCD_WIDTH-5,LCD_HEIGHT-25,
                     HGRAPHMODE_WHITE,screen);
        if (!(SymPtr->flags.bits.folder)) {
          sprintf(buff,"%u␣o.",*(unsigned short *)data);
45        hl_drawStr(hmenu->font,LCD_WIDTH/2+15,35,buff,FALSE,FALSE,screen);
        }
        hl_drawStr(hmenu->font,LCD_WIDTH/2+15,25,ext,FALSE,FALSE,screen);
      }
    }
```

Chapter 6

Configuration

The HibLib library needs some configuration information before begining the parsing of a text.

6.1 The structure

The function which needs some configuration information have in parameter `h_Config hcfg`. The structure to pass in parameter is the folowing one :

```
typedef struct {
  h_Lang lang;
  h_Font * font_msg;
  unsigned short speed_scroll;
  unsigned short speed_key;
} h_Config;
```

Then, for every function which needs some configuation information, you will pass in parameter the pointer of this structure. The structure would be filled before using a function of HibLib. In most cases, you don't need to change its values during using HibLib.

`lang`: defines the texts which will be used in informative messages (see the folowing section 6.2 about the internationalisation)

`font_msg`: defines the font which will be used for every message (see section 4 about font)

`speed_scroll`: defines the speed of the scrolling: this feature not enabled yet

`speed_key`: defines the speed between two keys pressed (a good choice is about 30)

6.2 Internationnalization

This library needs some text for some informative messages. To be independant of the language, every text have been externalized, and aren't included in the library. So, to use this library, you have to provide the definition of every text. The definition of the texts are defined in the folowing structure:

```
typedef struct {
  const unsigned char * link;
  const unsigned char * no_link;
  const unsigned char * bad_link;
....
  const unsigned char * log;
  const char * (log_err[NB_HLOG]);
} h_Lang;
```

Some possible translation are provided in the header file `hiblang.h`. For exemple, here is the french version:

```
#define HIBDLL_TEXTS_FR  (h_Lang){                          \
  .link="lien",                                             \
  .no_link="pas de lien",                                   \
...
  .log="Log",                                               \
  .log_err= { "Memory error to decompress the picture",     \
              "Error in the pretty-print expression",       \
...
              "The file is not a picture",                  \
              "The TiOS Font is not found"                  \
            }                                               \
}
```

So, to use the HibLib library with the french texts, you will have to fill the `h_config` structure like that:

```
h_Config hcfg = { ...
                  .lang=HIBLIB_TEXTS_FR,
                  ...
                };
```