1. Intro

The functions of this library base on Curses. They feature (not only) easy-to-use screen functions powered by a resource file. An attractive outfit can be realized with relatively small source codes.

The library is in fact an extension to the already powerful curses fuctions, and does not attempt to replace all calls to curses functions. The programmer can always choose which functions to use. But if you do not want to use Curses, you don't have to. Simple programs can be written without a single Curses call, although you then don't have concepts available with Curses.

Menus, Forms, Multiple-Choice boxes and more are configurable via resource files. Changes in the program layout is sometimes possible without recompiling. Another advantage is that the program itself has access to data inside the resource file and may read configuration data from there. If used consequently, a program may choose to read all its texts from a file. It might subsequently run with interfaces in different languages merely by replacing the resource file.

To use this library, you will need knowledge in C. The more knowledge, the better. Knowledge in Curses programming is useful, but as stated above, not a must. You should at least know what Curses is.

The library was originally implemented on SCO, and this is where it works best. It has now also been ported to Linux, HP/UX and Solaris, where it is more or less stable.

2. Compilation

2.1.... of the Library

At first, the library must be compiled. You will find a Makefile to help you with this task. Inside the Makefile, you will find lines with appropriate definitions for the Operating System and the compiler. Uncomment the definition for your system, and set the compiler. Once you have confirmed the functions are running, you may wish to set the FLAGS variable to '-O' for Optimization. If compilation fails, you might need to add a machine definition (email me if you succeed).

The Makefile should generate the resource compiler and two libraries, libresource.a and libnopanel.a. The second one is only needed if you do not already have a curses library on your system.

2.2.... of Applications

All modules which use functions of this library should include the following #include-Files:

```
#include <curses.h>
#ifdef NOPANEL
#  include <nopanel.h>
#else
#  include <panel.h>
#endif
#include <resource.h>
```

These directives will load all necessary definitions. Look below ("Systems lacking a Panel-Library") for a description of the meaning of the #ifdef. On compilation of each module, you'll have to add "-I<ResLibPath>" to the compiler command, where ResLibPath is the path to the directory where all the files of the Resource library resist (where you have compiled the library). On linking of the program, you must again specify the pathname in "-L<ResLibPath>" plus the libraries "-lresource" "-lpanel" and "-lcurses" in this order. For example, the example program "bemogrep" can be compiled and linked (on my home system) with

```
gcc -o bemogrep bemogrep.c -I/home/frank/resource -L/home/frank/resource \
          -lresource -lpanel -lcurses
```

2.3.Systems lacking a Panel Library

Many systems do have the Curses library, but not the also required Panel library. In order to get Resource programs to work on these systems, you have also got a replacement library. It contains all Panel functions, but is not intended to replace existing libraries. These functions are not optimized, and any original package should be substantially faster. But as long as you run the program on local terminaly, this should not be noticed.

This minimum replacement set is called 'nopanel', in order to be distinguishable from the real library. Of course, you need to tell your models to use the replacement functions instead of the real ones if necessary. But this is really easy. At first, you must define NOPANEL in the Resource Makefile and recompile the library from scratch (make new).

If you have used the statements given in the last chapter, all of your modules are already prepared to use NOPANEL functions. There, a #ifdef-Directive is used to load either the original functions or the fakes. You now only need to define the NOPANEL variable in your Makefile. And, you will have to replace the '-lpanel' link command by '-lnopanel'. And don't forget to recompile your files as well.

WARNING: Compiling some of the files with NOPANEL defined and some without will cause unpredictable problems (hopefully, your compiler or linker will scream). Although both libraries define the same functions, they are NOT compatible.

If your system does provide you with a panel library, and you experience problems with screen refresh (parts of the screen are not correctly refreshed), you might also want to try the nopanel replacement. Chances are that it won't get better (usually, Curses is the real problem), but you never know.

3. The Resource File

The resource file is basically designed to be simple. All resources are defined serially. Everywhere, comments may be begun with '#'. If a '#' is encountered while reading the resource file, the rest of the current line is ignored. There are also conditional comments. You can use conditional comments to define resources differently in different contexts. On startup, an application may initialize a string variable named accpref, which means "ACCeptable PREFixes".

A conditional comment starts with a '^', followed by one or more prefixes. Each prefix is one character in length. Each element of the accpref string is an acceptable prefix. If the '^' is followed by one or more of the acceptable prefixes, the rest of the line will be read. If none of the characters after '^' is an acceptable prefix, the line is ignored.

Example: The program initializes the string with strcpy (accpref, "abc"); and the resource file contains the following lines:

```
^a Line 1
^c Line 2
Line 3
^z Line 4
# Line 5
^fb Line 6
```

Line 1,2,3 and 6 will be read. Line 1 and 2 have an acceptable prefix. Line 3 is neither comment nor conditional comment. The prefix on line 4 is not acceptable. Line 5 is an unconditional comment. Line 6 also contains 'b' as acceptable prefix. Later, you will also learn how complete resources, and not only single lines, may be conditionally ignored.

Everything which is not a comment is a declaration or part of a resource. A resource is declared by the resource header:

```
[<Name-of-Resource> : <Type-of-Resource> : <HelpContext>]
```

The most important part of this declaration is the type of the resource. The name of the resource is just an alphanumeric string, relevant only to the application. The HelpContext is an optional reference (in fact, the ':' and the HelpContext may be omitted) to another resource of type "Text" by the name of "HelpContext". As of now, this value is only used for resourcetypes "Select" and "Form", that is, for multiple-choice boxes and forms, in order to get contextsensitive help with the F1 key.

Directly after the initial bracket may again be a '^' followed by acceptable prefixes. If one of the characters is found in the accpref string, the resource will be loaded, if not, the complete resource will be ignored (no need to conditionally comment each line).

The description (the body) of a resource of course depends on its type. The rest of this chapter will try to introduce you to the different resource types. There are the following:

3.1.Resource

This resource type is simply a reference to another resourcefile. This resource must have a name, although it is ignored (name it 'Dummy' or something). Its body is exactly one line with the absolute pathname of the to-be-loaded resource file. Example:

```
[Dummyname : Resource]
/usr/local/lib/application/german.res
```

All recursive loads are evaluated, although no check for loops are performed (if A loads B and B loads A, your system will be hanged).

3.2.Parameter

This is just some sort of parameter. It is not evaluated by the resource functions, but is passed to the application program. This is the interface for the application to gain access to the resource file. The body of a Parameter resource completely depends on the application program. This feature is described below.

3.3.Message

A Message is exactly one line of text. It is normally used to describe short messages to the user, like in message boxes (described below). If possible, all one-liners in a program should be defined as messages in order to locate them outside of code. This will make stylistic changes much easier. The following example uses a message and conditional comments to define a "gentleman's" version and a "hey-shut-up" message:

```
[FileNotExist : Message]
^a This file does not exist. Please try again.
^b Hey man, you are such a bore. Come to your senses and try again!
```

The characters '[', '\',]', '^' and ':' must be escaped in messages with a backslash '\'.

3.4.Tvector

This is a multi-line message (that is, a vector of messages). The first line of the body must contain the integer number of text lines. These text lines follow afterwards. Since empty lines are ignored in a resource file, a '@' as the first character on a line is interpreted as an empty string. Any text following the '@' is ignored.

```
[<Name-of-Vector> : Tvector]
<Number-of-Lines>
<Line 1>
<Line 2> ...
```

You must be careful when using conditional comments in a Tvector, because the lines which are commented out by a conditional comment may not be counted by the *<Number-of-Lines>*. Either in all contexts exactly the same number of lines remain, or *<Number-of-Lines>* must be itself conditionally defined different for each context.

Again, you must escape the characters '[', '\', ']', '^' and ':'. If you want a text line to contain '@' as its first character, you must escape it as well (so it becomes the second character).

3.5.Text

A text resource is a text of unknown length and is usually used for help texts (since the only way to access a text resource is a function to actually display it). The first line of the resource body contains the title of the text (which will be presented as a headline). The layout of the text is not relevant, since all linefeeds are ignored. When being displayed, the line length is automatically adjusted to the actual window/screen width.

A text may contain four different directives to change the style of the text. '\\b' switches to block alignment (default), "\\c" to center alignment, "\\l" and "\\r" to left and right alignment, respectively. All these four directives cause a line break (thus, if you want to begin a new paragraph, use dummy text style switches, like "\\b\\b").

A text ends at the next resource header. Again, the characters '[', '\', ']', '^' and ':' must be escaped with a backslash. Note: The actual alignment directives have only one backslash. In the above description, this backslash is already escaped, resulting in the double backslash.

3.6.Select

Now we're getting to the more complicated and even more useful resources, since they also feature an interface. The select resource defines a select box, which allows the user to choose one of many items.

```
[<Name-of-Resource> : Select : <HelpContext>]
<Number of Items>
<YPos> <XPos> <Title>
<ID0> <Item 0>
<ID1> <Item 1>
...
```

<Number of items> defines the actual number of items to follow. *<YPos>*/*<XPos>* are the screen coordinates where the select box will appear. Since a frame is drawn around the box, *<YPos>* must be larger or equal than 1 (3, if it has a title), and *<XPos>* must be larger or equal than 2. Both may also have the values of "-1" (left(upper)aligned), "-2" (right(down)aligned) or "-3" (centered). The size of a select box is calculated by the length and number of its items.

<Title> is the title the Select box will have. If it shall not have a title, use "@". Select boxes are also used within a menu. A box that is used within a menu must have neither coordinates nor title. In this case, the second line of the body is just "@".

Below, the box's items are declared. *<Item n>* is a text string that appears in the box. Each item has an *ID* which is used to identify the item. These *IDs* will be returned to the application on display of the box. *IDs* should therefore be unique within a box and within a complete menu. *ID* values of less than 100 are used for internal purposes; actual *IDs* should be greater or equal than 100.

If a user presses the Help Key (F1) while choosing from a select box, the text resource by the name of *<HelpContext>* is displayed on the screen.

3.7.Menu

A menu is meant to be the title line of a program's screen layout. There, a couple of menu items are presented. Each of the menu items has again a list of possible choices. A menu item can either be a plain text (without a submenu) or a select box, providing the mentioned choice. A menu is declared as follows:

```
[<Name-of-Resource> : Menu : <HelpContext>]
<Number of Items>
<YPos0> <XPos0> <ID0> <Text0> : <SelectBoxName0>
<YPos1> <XPos1> <ID1> <Text1> : <SelectBoxName1>
...
```

The first line defines the number of menu items, and each following line describes one item. *<YPos>*/*<XPos>* are required and place the menu on the screen. Vertical positions greater than 0 are accepted, although they may result in strange screen layouts when browsing the menu. Horizontal positons must be greater or equal than 1, and the length of the menu name plus its *<XPos>* should be less the screen size.

<Text n> is the menu name that is displayed on the screen. Each menu's *<XPos>* should depend on the previous item's position and its length.

A menu item must have either an associated selectbox, referred to by its resource name, or an *ID*. In the later case, the menu item does not have a submenu; and the ':' and the *<SelectBoxName>* must be omitted. If a selectbox is specified, the *ID* must be omitted. This specific selectbox should have neither title nor a screen position (which is calculated according to the menu item's location).

If the user selects a menu item without a submenu, its *ID* is returned. If the user selects a submenu item from a select box, the *ID* of this specific selectbox item is returned. Therefore, *IDs* should be unique within a complete menu.

3.8.Form

A form is by far the most complicated resource. It describes a complete screen form, in which the application may enter data, which is then changed by the user and again read out by the application. A complete user interface can be imagined using only forms and menus. A form resource is described as:

```
[<Name-of-Resource> : Form : <HelpContext>]
<Number of Items> <YSize> <XSize> <YPos> <XPos>
<Title>
<TYpos0> <TXpos0> <Text0> : <ID0> <FYpos0> <FXpos0> <Length0> <Flags0>
...
```

Again, the number of items in the form is mentioned first. Following is the screen size and position of the form. As for a select box, the values "-1", "-2", "-3" may be used for the positions as left/right/center alignment, respectively. The form's size must be explicitely specified. When

specifying explicit positions, remember that YSize>=1 (3, if the form does have a title) and XSize>=2. If a form should not have a title, set the second line to '@'.

After this general information follow one-liners for the definition of each of the form's element. Each element consists of two parts, the element's title and the element's input field. The first part is defined before the ':', the input field's definition is located after the ':'. An element does not need both parts. An element might only be a title (or, in this case, a freely locatable text) or an input field without a title (if no description of the field's intention is necessary, or mentioned elsewhere). In the first case, omit the ':' and the rest of the line. In the second case, omit everything before the ':' (which must still be present!).

The title of an element has the position *<TYpos> <TXpos>*. This position is relative to the form's positon. 0/0 is the upper left corner inside the form. An input field begins at *<FYpos> <FXpos>* and is of length *<Length>*. Each input field must have a unique *ID* (needs only to be unique within one form) associated to it (use only *ID* values greater than or equal to 100). This *ID* is used by the application to refer to the field. The field's *IDs* do not need to be ordered. But what matters is the ordering of lines within the resource description. This is the order in which the cursor moves from field to field. You can therefore change the course of the cursor merely by rearranging the lines.

<Flags> is a series of single flags in direct sequence (without spaces inbetween). There are the following flags (shown here in quotes, the quotes do NOT belong to the flag):

- 'A' This field accepts alphabetic characters (uppercase and lowercase)
- 'N' This field accepts digits ('0' - '9')
- 'C' Special characters like '*' or '.' etc. will be accepted
- 'S' The space character will be accepted
- 'U' All characters will be translated to uppercase on input (it will thus become impossible to enter lowercase characters)
- 'D' All characters will be translated to lowercase on input (it will thus become impossible to enter uppercase characters)
- '-' This field is not active. It will not be highlighted, and the user may not enter data in this field. The application can print data into this field, to show fixed values that are of interest to the user.
- 'l' downlight the field. The field will not be hightlighted, but the user can still enter data.
- 'F' activates function keys. The application program will be notified of any function key the user has pressed while editing data in this field. In order to make function keys active throughout the form, you have to repeat the 'F' flag for every non-inactive field. 'F1' as the help key will still be processed internally, as well as all other function keys that have aliases, macros or functions associated with them.
- '!L' activate field. The application program will be notified whenever the user leaves this field.

Among the flags may also be a quoted text string with a length of less than *<Length>*. This sets a default value for the field. The use of default values is not implemented consequentially, and you may run into problems when trying to mix default values with other features.

Example: "AS" might be a sensible value for a name field. The user can then enter his full name, separated by spaces. If you want to allow middle initials (as in. "Alfred E. Neumann", you must add 'C' to allow the '.' character).

3.9. Unknown Resource

All resources that are not of the above types are passed to the application's function *LoadExternResource()*. Through this feature, each application may define its own resource types. Of course, the function is itself responsible for correctly reading in the types' data.

Sadly, also all resources where the resource type has a typing error will be passed to this function. The application should therefore report all unknown resources (write them to an error log during the test phase). And usually, a missing resource will be noticed.

4. Basic Functions

This chapter will introduce some basic functions that do not produce actual screen output. They are available to the application to initialize and finish the curses functions, to query certain information and to read values from resource files.

4.1. General functions

- `int StartCurses (char *filename, char *intres, int monoflag)`
  This function initializes the screen functions and all necessary structures. It needs to be called once before any curses-oriented screen output is attempted. This function does all curses initialization, so no other call to *initscr()* is needed. If *filename* is non-NULL, the resource file by this name is read.

  Resources can also be read in from memory. *intres* may point to a string containing the equivalent of a resource file. The concept of 'compiling' resources will be introduced later. Both a filename and an internal resource will be evaluated.

  Usually, the library tries to initialize a color terminal and set some colors. The terminal is queried if it supports color, and if it doesn't, color use is suppressed. Unfortunately, some terminal implementations (especially in X-Windows) report that they are capable of colors, but then display only black and white. This may result in white text on white background. By setting *monoflag* to 1, color use is not attempted to prevent this behaviour.

  The function returns a TRUE value on success and false on failure. As of now, the single reason for failure is a reading failure on filename, but later implementations may check for more cases (the curses initialization may also fail on line-oriented terminals, but this results in the exit of the application).

  After this function has been called, the global variables LINES and COLS are filled with screen heigth and width. The application should check if these values are large enough to display all of the application's information and fail if they aren't. Of course, an ideal application will adjust to different screen sizes.

- `void EndCurses (void)`
  Ends all curses actions, clears the screen (hopefully) and frees all resources. Must be called before the application terminates. After this function, no more curses functions may be called. The screen may then again be initialized via *StartCurses()*.

- `int GetWinSize (int *lines, int *cols)`
  Checks for the size of the terminal screen. If the check succeeds, the screen heigth will be written to *lines*, its width is written to *cols*, and TRUE is returned. If the check fails, the value of *lines* and *cols* is undefined and FALSE is returned. This function is only useful when the application is run in a resizeable X window.

  The X utility "resize" is called to get the new values. You need to have this program in your path.

  If this function succeeds and the screen size has indeed changed, the application may ignore the change (producing a possibly scrambled output) or may readjust itself. In this case, the global variables LINES and COLS must be reset by the application, and *DoResize()* (see below) must be called.

- `int HaveToResize (void)`
  The above function stresses resources, because an external program must be spawned. This function serializes queries for screen size changes and makes sure this effort is done at most once in ten seconds. If this function returns TRUE, the application can be sure that screen size has changed, and that *GetWinSize()* can be called to acquire the new values.

- `int DoResize (int lines, int cols)`
  After the screen size has changed, and the application has readjusted itself for utilization of the new dimensions, this function must be called to reset the values of the physical terminal. Before this function has not been called, no output should be written to the screen. This function clears stdscr.

- `void ScreenToFile (FILE *file)`
  Writes all contents of the current screen, curscr, to *file*. Semigraphical characters are changed to standard characters as good as possible.

- `void RedrawScreen (void)`
  If the screen has become garbled for some reason (console message), this function will restore the screen by completely clearing and rewriting it. Since only a user will notice the garbled screen, there should always be some key combination by which this function is called.

- `int HaveABreak (void)`
  The resource functions install a signal handler for the SIGINT signal, which is usually sent by the CTRL-C key combination. The signal handler will present a window, querying the user if he really wants to cancel the action. This function returns TRUE if such an event has occured. The flag is automatically cleared whenever the application waits for some input (this means, there is no action to cancel).

  While doing lengthy actions, the application should check from time to time if this function becomes TRUE, and if it does, it may stop the current action. Once this function is TRUE, it will stay TRUE until either user input is awaited (with *InputChar()*, see below) or the flag is reset by the next function.

- `void BreakProcessed (void)`
  This function resets the break flag to FALSE. This two-function mechanism is presented so that the application can check the above function on different levels. Once the application has returned to top-level (e.g. to the menu loop), this function is called.

- `void DeInstallCtrlC (void)`
  Uninstalls the SIGINT signal handler and installs the ignore signal handler (so that CTRL-C will be ignored). The described window will not be presented to the user. The signal handler should always be uninstalled while the application is doing lengthy, uninterruptable actions. Calls to *HaveABreak()* are senseless.

- `void InstallCtrlC (void)`
  Reinstalls the signal handling routine and therefore reenables the checking for cancel requests via *HaveABreak()*.

- `char *calcstr (char *instring, char *outstring)`
  This function implements a small calculator. *instring* is primarily copied to *outstring*. An ampersand '&' begins a calculation. This calculation must be in prefix notation, like in &(+ 1 (* 2 3)). Only the four basic operations

'+', '*', '-' and '/' are implemented. Each operation may have only 2 operands. If *instring* is "The result is &(+ 1 1)", then *outstring* will become "The result is 2". The function returns a pointer to *outstring*.

The above is not quite thrilling. But the function gets useful with the utilization of application-defined variables. A variable is an alphanumeric string enclosed by dollar signs (as in "$PAGE$"). There are only integer variables. Whenever this function encounters a variable, an external, application-defined function is called to evaluate the variable string and return its value.

This process may be used to dynamically process strings from a resource file. The process is not easy, but at least possible. Example: The application reads in a string from the resource file "The current date is &(* 1 DAY).&(* 1 $MONTH$).&(* 1 $YEAR$)"; this function converts it to "The current date is 1.1.2001".

- `RESOURCE *LoadResource (char *file, char *intres)`
  Reads in additional resources. If *file* is non-NULL, data is read from the file by this name, else the internal resource specified by *intres* is evaluated. After this function has been called, the resources must be added to the already existing resources with the following function. This function returns NULL if *file* is unreadable or empty or the internal resource is empty.

- `void AddMoreResources (RESOURCE *res)`
  Adds the resources specified by *res* to the already existing resources. Resources in *res* overwrite previous definitions. This function is only used in connection with the above function. A typical call is as follows:

```
AddMoreResources (LoadResource (filename, NULL));
```

4.2.File Functions

The following functions provide an interface to read data from resource files. Usually, they are only used internally, but maybe they can be put to use elsewhere, too. All functions are prepared to take data either from a file or from memory.

- `int FGETCC (FILE *file, char **ptr)`
  Reads a single character. If *file* is an open file pointer, then it is read from the file. If *file*=NULL, then it is read from the memory pointer **ptr*. This is a macro.

- `int FEOFC (FILE *file, char **ptr)`
  Tests the end-of-file condition for either the stream *file* or the memory pointer *ptr*. This is a macro.

- `int UNGETCC (int c, FILE *file, char **ptr)`
  This function ungets the character *c*, this means, *c* will become the next character to be read. In the case of a memory pointer, only the pointer will be decreased by 1. Since usually only previously read characters are UNGETCC()'d, this difference will not be noticed. This is a macro.

- `char *FGETSC (FILE *file, char **ptr, char *string)`
  Reads a complete string from the *file* or from the memory position and copies it to *string*. At most 253 characters are read. string will then be finished with a '\n' and a nullbyte. The function respects all possible end-of-line styles (DOS, MAC and Unix-style) on input, but will copy just one '\n' to string in all cases.

The above functions are not too bright, they just mirror the standard functions and extend them by the feature of being able to read from memory. The following functions are better specialized for reading resource files. Comments and conditional comments are ignored. Applications using these functions do not need to worry about comments, they simply won't see them. The followwing functions use internal buffering, and they must therefore not be mixed with standard functions, and also not be mixed with the above functions. The only exceptions is FEOFC(), which still must be used to test the end of file. A file must be read from opening to the end-of-file condition by these functions.

- `int freadi (FILE *file, char **ptr)`
  Reads an integer number from the file or from memory. Comments preceding the number are ignored. If a '%' is encountered, a calculation in the sense of *calcstr()* is performed (in this case, the '%' replaces '&'). Empty lines, spaces and tabulators preceding the number are ignored. Spaces, tabulators and up to one line break after the number are ignored. The function returns the read number or the result of the calculation or -1 upon failure. If -1 is a valid value for the expected number, you cannot check whether an error has occured or not.

- `char *freads (FILE *file, char **ptr)`
  Reads a string up to the end-of-line from the file or from memory. Comments and empty lines preceding the string are ignored. If the string contains calculations introduced by '%', they are evaluated. The function respects all line termination styles. The end-of-line character is not transferred to the string. The function then allocates memory via *malloc()*, copies the string to that location and returns a pointer to it. The memory must later be freed by *free()*. This function also evaluates escaped characters.

- `char *freadc (FILE *file, char **ptr)`
  Reads a single character from the file or from memory. Empty lines and comments are ignored. At the end of an non-empty line, the end-of-line character is returned. This function does not evaluate escaped characters (the backslash will not be processed).

- `void fungetc (int c)`
  Writes the character *c* back into an internal buffer. It doesn't matter if the character has beed read from a file or from memory, or if it is just invented by the application. It will be returned by the next call to *freadc()*, or evaluated by *freadi()* or *freads()*.

4.3.External functions

The following functions are not defined by the resource library. Each application program must define these functions, if only empty. They are called when reading resource files or evaluating calculations.

- `int getcval (char *string)`
  This function is called by *calcstr()* to evaluate application variables. String points to an alphanumeric string enclosed in dollar signs. This function must compare the string to variable names and return the appropriate value. Unknown variables may be reported to an error log during the application's test phase or ignored.

  Possible variables include LINES and COLS for screen heigth and width.

- `void LoadExternResource (FILE *file, char **ptr, char *rname, char *rtype)`
  Resource files may define application-specific resources. Whenever the resource file reader encounters a resource unknown to the library, this function is called. *rname* points to the name of the resource, and *rtype* to the type of the resource. If no name or type was declared in the ressource file, they might be empty or NULL. Both strings will be deleted on return of this function and must therefore be copied if needed by the application. The file/memory pointer is positioned on the first character of the resource's body.

  The function must use the above declared functions for reading from file or memory. *file* and *ptr* may be passed directly to them. This function must now determine the length of the resource and read it. If the resource's length is unknown, the function should check each line for an opening bracket '[', *fungetc()* this character and then return. The function should also check whether the end of file is reached (*FEOFC()*).

  During the testing of the application, you might want to report unknown resources to an error log.

5. Structures

This chapter will introduce to you the data structures used by the library. It will be seldomly necessary for the application to access these structures by hand. They are mentioned because the application must define some of them, and you should know what you're defining, shouldn't you?

All structures are filled when the resource file is read. Creation of own resources at runtime is not encouraged, but possible (like dynamic select boxes, where the items are only known at runtime).

- RESOURCE - resource of arbitrary type
- SBOX - resource of type select box
- MENU - resource of type menu
- TEXT - resource of type text
  This structure does not hold the text itself. It is just a reference to the text. If the text comes from a resource file, then the filename and the file position, pointing to the first line of the body of the resource are stored, if the text comes from memory, the equivalent memory position is stored.
- FORM - resource of type form
  This structure holds all the definitons necessary for a form, only it does not have memory allocated for all the input fields. A form is never referenced by this structure, but only by the next.
- FINFO
  This structure is an extension of the previous ones, it holds an initialized form. Memory is allocated for the input fields, and the screen windows are also initialized.
- PCBAR
  This structure describes a "bar", which can be used to display progress information from 0-100%.

The next two structures are not defined by the library but by curses itself. They shall only be mentioned here shortly; you will find more extensive information in the curses manual or manpages. The application programmer is encouraged to make himself familiar with them, since they are needed quite frequently.

- WINDOW
  This structure is nothing more than just a section of screen. A window is initialized by the curses function *newwin()*. A window must always be inside the screen (may not extend beyond its borders). Concurrent windows may overwrite each other.
- PANEL
  This is a window with added depth. A panel may overlap with other panels. The application programmer should never use single windows unless they have been associated to a panel by *new_panel()*. However, a panel is nothing in itself. Panel functions are only responsible for screen update. Data must still be directly written to the window.

6. Keyboard Functions

These functions are the interface for keyboard input. No other functions should be used.

- `int InputChar (void)`
  Reads a character from the keyboard. If the keyboard buffer is empty, the function waits until the next key is pressed. The character is not echoed. Function keys are detected, and the appropriate value is returned. This function implements lots of checks for different actions that can be set and unset with the following functions. There is yet no possibility to poll the keyboard non-blocking.

  If the user presses the help key (F1), the current help context (see later, or in the resource file structure explanation) is evaluated, and the text is displayed on-screen. Upon return from the display of the help text, the function waits for another key.

- `void SetKeyAlias (int readkey, int writekey)`
  Redefines a key. Whenever the user now presses *readkey*, the above function returns *writekey*. *InputChar()* will also test if another alias exists for *writekey*. Be careful when redefining keys not to create a loop, since it will cause an infinite loop. If more than one alias exist for readkey, the last one is used.

- `void DelKeyAlias (int readkey)`
  Cancels a defined alias for *readkey*. If more than one alias exist for *readkey*, only the last one is canceled, so that the previous one is reactivated.

- `void SetKeyHook (int readkey, void (*function) (void));`
  Defines *readkey* to be a hotkey. Whenever *InputChar()* now encounters *readkey*, function will be called. This function may do everything it wants to, including screen update or further keyboard inputs. Upon return from this function, another key will be read. If more than one function is bound to the same hotkey, only the last funtion will be called. A good example is *DisplayHelp()*, which is by default bound to the F1 key.

- `void DelKeyHook (int readkey)`
  Cancels a previously defined function for *readkey*. If more than one function is bound to *readkey*, only the last one is canceled, so that the previous one is reactivated.

- `void SetKeyMacro (int readkey, int *macro)`
  Defines a keyboard macro for *readkey*. macro is a null-terminated array of key codes. Whenever *readkey* is now read, *InputChar()* will simulate the sequential input of all values in macro. The array must not contain *readkey* itself, since this would result in an infinite loop. If the length of macro is 1, the behaviour of this function equals *SetKeyAlias()*. If more than one macro is assigned to readkey, only the last one will be used.

- `void DelKeyMacro (int readkey)`
  Cancels a macro for *readkey*. If more than one macro is associated to *readkey*, only the last one is canceled, so that the previous one is reactivated.

All these keyboard functions influence each other. If a macro contains characters for which aliases are defined, these aliases will be used. Even hotkey functions may be called by a macro or by a key alias. If used excessively, this will soon lead to infinite loops.

A macro has highest priority before the alias and the function. If more than one action is defined for the same key, only the macro will be used.

7. Resource Functions

These functions access or are related to the data in the resource file.

7.1.Basic functions

The first two functions retrieve data from the resource file. The first one retrieves a single character string, the second one an array of strings (Tvector). They are usually used to get short texts for screen output. A consequential application will place all text information into the resource file and have no hardcoded texts at all. This hopefully eases future changes.

- `char *GetMsgResource (char *name)`
  Retrieves the resource of type **Message** by the name of *name* from the resource file and returns a pointer to the text contained in this message. It points to the original data. If the application intends to change the text, it should copy. If the resource is not found, NULL is returned.

- `char **GetTvecResource (char *name)`
  Retrieves the resource of type **Tvector** by the name of *name* from the resource file and returns a pointer to the array of strings containing the individual elements. Before changing any of them, they must be copied, too. A null pointer marks the end of the array. If the resource is not found, NULL is returned. A legal way (providing no error occurs) to print a Tvector would be:

```
array = GetTvecResource ("ToBePrinted");
while (*array)
    fprintf (file, "%s\n", *array++);
```

- `void SetHelpContext (char *helpres)`
  This function pushes the resource by the name of *helpres* onto the help stack. It does not copy the string, so it should be either a static string or it must be made sure it doesn't change until it is pulled from the stack by the next function.

  Now whenever the application waits for user input and the user presses the help key, the **Text** resource by the name of *helpres* will be looked up, and displayed as help text on the screen. Whenever the application queries user input with *InputChar()*, an appropriate help context should be set. If the application does not want to provide help, a help context of NULL may be set. This makes sure the user doesn't get completely wrong information. Instead, a window will pop up to tell the user that help is not available.

  Within select boxes, menus and forms, the help context will be determined by the resource (from which the help context part is here evaluated).

- `void DelHelpContext (void)`
  Pops the most recent help context from the help context stack, and the previously set help will again become actual. The most straightforward approach to presenting context sensitive help is as follows:

```
SetHelpContext     ("WaitingForAKey");
result = InputChar ();
DelHelpContext     ();
```

7.2.Simple resource functions

These are simple functions needing only one call to do the job

- `void InfoUser (char *title, char *text)`
  This opens up a neat little window with an 'i' for information on the left, the *title* written in the title bar and the *text* written to the right of the 'i'. If text is longer than the window's width, it will be automatically wrapped, so don't worry about that. text should be shorter than 150 characters and may not contain any line feeds or carriage returns. The function waits for the user to press the return key, closes the window and therefore leaves

the screen untouched. Remember that *title* and *text* should come from the resource file. An advanced function call could look like

```
if (error)
    InfoUser (GetMsgResource ("ErrorTitle"),
              GetTvecResource("ErrorDescription")[error]);
```

This will not only inform the user that an error has occured, but it will also present an error-specific description. Of course it is also possible to construct the texts at runtime, providing maybe an error message that was returned by another function.

- `void DisplayText (char *textres)`
  Looks up the resource of type **Text** and name *textres* from the resource file, reads the text, formats the text (alignment as specified in the description of the resource file's structure), opens up a window and displays the text in this window. The user may browse through the text. The function waits for the user to hit the return key, closes the window and returns.

- `int DoSbox (char *sbres)`
  Looks up the resource of type **Select** by the name of *sbres* from the resource file, opens up a window, fills the window with the data items of the select box and lets the user choose one of the items. If the user hits the return key, the ID of the chosen item is returned. If the user cancels the box by pressing the escape key, SB_ESCAPE is returned. Upon error (select box not found), the function returns -1.

- ```
  int DynSelectBox (int anz,
                    char *title, char *hc,
                    char *ppage, char *npage,
                    char **items)
  ```
  Creates a dynamic select box with data not retrieved from the resource file but created at runtime by the application. *anz* is the number if items; these items are contained as strings in the *items* array. *title* is the text that will appear in the title of the select box and may be null if the box shouldn't have a title. *hc* is the help context (the text resource which will be displayed if the user hits the help key).

  If there are more items that can fit on the screen (the function will display a box with at most LINES-10 data items, that is, if *anz*>LINES-10), the function will offer the user the possibility to browse through multiple pages of data. *ppage* is the text that will displayed as "previous page", and *npage* as "next page"; this allows the application to present some fancier strings here. If *ppage*/*npage* is NULL, then the user will not be able to browse backwards/forwards, respectively.

  Remember that *title*, *ppage* and *npage* are text strings that will appear as given (instead of being looked up from the resource file). Therefore these text are usually fetched from a **Message** resource by the application itself.

  If an error occurs while creating the dynamic select box (no memory or something), the function returns -1. If the user cancels the selection by pressing escape, the function returns SB_ESCAPE. If the user selects a menu item, the function returns 1000+index (where index is from 0 thru *anz*-1), with *items*[index] being the item selected.

- `void InitMenu (char *menures)`
  Initializes the **Menu** resource by the name of *menures*. The menu items are printed onto stdscr, and a line is drawn on the first line of the screen. This function only works sensibly if all menu items are on line 0. If an application uses multiple menus, it does not need to un-initialize a menu. The application can just call this function repeatedly with different resources. The last initialized menu becomes the current one.

- `int DoMenu (char *menures)`
  Opens the **Menu** resource by the name of *menures*. The menu should be previously initialized by the previous function. The user may then browse the menus and submenus. If the user hits the return key, the ID of the chosen item is returned. If the user canceled the menu by pressing the escape key, or if an error occured, -1 is returned.
  ```

7.3. Usage of Bars

Bars can be used to illustrate the progess of a process. While doing an extensive action, the application can show a bar indicating how many percents of the action is completed. This will not only tell the user more or less accurately the time he still has to wait, it is also accepted as a life sign of the application (never let the user think the app might have crashed).

- `PCBAR *InitBar (int maxval, int ypos, int xpos, int length)`
  Initializes a bar at the screen position *ypos*/*xpos*. The bar will have the specified length. The function will also draw a frame around the bar, which causes additional 1 characters in height and 2 characters in width. *ypos* and *xpos* may also hold the values -1/-2/-3 for left/right/center alignment, respectively.

  maxval is an integer number indicating the 100% value and must not be equal to or less than 0.

- `void SetBar (PCBAR *bar, int value, int maxval)`
  Sets the value of the bar to the specified value. *bar* must have been previously initialized by the previous function. The bar will now show the percentage of *value* against maxval. If the *maxval* given in this function call is 0, the maxval from the initialization is usen. If *maxval* is positive, this maximum value is reset to the new value. This allows the application to dynamically adjust the bar to new conditions (although you should explain to the user somewhere why the bar may run backwards).

- `void DelBar (PCBAR *bar)`
  Deletes the bar from memory and from the screen.

7.4. Usage of Forms

There are plenty of functions available to manipulate forms. This should be understandable, since their job is quite complex (think of the resource file definition ...).

- `FINFO *InitForm (char *formres)`
  This function loads and initializes the resource of type **Form** by the name of *formres*. Memory is allocated and everything is prepared. But the form is not yet displayed on screen. To display it, you still have to call *DisplayForm()*. You can use multiple copies of the same form by repeatedly calling *InitForm()* on the same resource.

- `void DisplayForm (FINFO *info)`
  Displays the form. This call is ignored if *info* is already visible.

- `void HideForm (FINFO *info)`
  Hides the form. This call is ignored if *info* is already hidden. This affects only the form's visibility, but not its behaviour. You may still retrieve data from the form, put data into the form etc.

- `void PutForm (FINFO *info, int resid, char *text)`
  Writes *text* into the field by the ID of *resid* to the initialized form *info*. The string will be cut if it is too long to fit into the field. If *text* is shorter than the field, it is filled up with spaces, and if *text* is NULL, the field will be emptied. The field is assigned only a copy of *text*, so that the application may happlily overwrite the string.

- `char *GetForm (FINFO *info, int resid)`
  Searches the form *info* for the field with the ID of *resid* and returns its raw data. If no field exists by the specified ID, NULL is returned. If successful, the returned string points to the field's data and must be copied before changing it. This string is also filled up with spaces up to the field's length. You therefore shouldn't use functions like *strcmp()* on the return value of this function. The *ShortString()* function is provided by the Resource Library to get rid of all the padding spaces. If you only want to see if the field is empty, you can use

  ```
  IsStringEmpty (GetForm (form, resid));
  ```

- `void PutFormDefaults (FINFO *info)`
  In the resource file, you could set a default value for each field. The call of this function sets all fields in the form *info* to its default value. If no default value was defined for a field, it will be emptied.

- `int QueryForm (FINFO *info)`
  This function allows the user to modify data in the fields of form *info*. If the form is not visible, it will be made visible upon entry of this function, and will be hidden again upon leave of it. If the user cancels entering data, IT_ESCAPE is returned. If the user presses the return key, IT_OK is returned. If the user has canceled the form by pressing escape, the form still holds the values the user has entered until his pressing of escape.

  If Cursor leaves an active field (flag '!L' in the resource file), the function returns *-resid*, where *resid* is the ID of the left field. In this case, this function **must** be called again. Whenever a form has active fields, this function must be called from the inside of a loop, that may only terminate when it gets something else than *-resid*. This state is referred to as a "query loop".

  If the user hits a function key inside a field with active function keys (flag 'F' in the resource file), KEY_F(*n*) is returned, with *n* being the number of the function key. This is not true for function keys that have aliases, macros or functions associated to them. These will be processed first.

  This is simple code that only accepts a form if the user has entered some data in field 1000:

```
end = data = 0;
do {
    switch (QueryForm (form)) {
          case IT_ESCAPE:   /* break out with failure */
                end = -1;
                break;
          case IT_OK:       /* if data is ok, then break out of loop */
                if (data)   /* else ignore                           */
                      end = 1;
                break;
          case -1000:       /* the user may have changed this field  */
                if (IsStringEmpty (GetForm (form, 1000)))
                      data = 0;   /* field is empty */
                else
                      data = 1;   /* you might want to validate data */
                break;
          default:          /* we shouldn't get here */
                break;
    } /* switch */
} /* do */
while (end == 0);
```

- `void MoveForm (FINFO *info, int ypos, int xpos)`
  Moves the form, visible or not, to the position *ypos*/*xpos*. This is an absolute postion and may **not** hold the values -1/-2/-3.

- `void SetFormCursor (FINFO *form, int resid)`
  Sets the input cursor to the field with the ID *resid*. The call is ignored if the form does not have a field by this ID, or if the field is inactive (flag '-' in the resource file).

- `int GetFormCursor (FINFO *form)`
  Returns the ID of the field in which the cursor is presently located. The return value of this function is not valid if a *QueryForm()* call has just returned because of the change of an active field (any *-resid* was returned). It is valid in all other contexts of this loop, as well as anywhere outside a query loop.

- `void DeleteForm (FINFO *form)`
  Deletes the form from screen and from memory.

All these dry descriptions may not have revealed all the power that comes with the use of forms. Especially active fields, or activated function keys (both features didn't exist in earlier versions of the

library), imply lots of features. In many fields only discrete values may be entered. You can now check each time when the user has changed data in such a field if the data is valid. You might not want to allow the user to leave a field with illegal data. Or, if the user presses some function key, you could present to him a select box with valid input values (createable at runtime with *DynSelectBox()*), from which he may choose an item.

Forms are also recyclable. After a form is initialized, you can always pop it up anytime, let the user enter data, and then hide it again with *HideForm()* until the next time. Remember, if you hide and redisplay it, it will still hold all the data previously entered. You could easily create a form holding some of the application's vital parameters that can be summoned by a hotkey (see *SetKeyHook()*).

You can also create forms completely without user-changeable data just to display information on the screen, since handling of a form is much more simple than programming curses windows by hand.

Screen Functions

The following functions exist to simplify direct access of the screen (without resource functions). They operate directly on the WINDOW data structure of the curses library. Even if these functions use the WINDOW structure, keep in mind that you may never use WINDOWS without a panel associated to it. Using non-paneled WINDOWS may result in undefined behaviour of screen output.

- `void wfill (WINDOW *win)`
  Fills the window *win* with spaces, using the attributes set for this window (like background color, see *wattrset()* in the curses manual)

- `void wfillrgn (WINDOW *win, int y1, int x2, int y2, int x2)`
  Fills only the part of the window *win* that is specified by the coordinates, inclusive. The upper left corner of a window is 0/0.

- `void wbox (WINDOW *win, int y1, int x1, int y2, int x2)`
  Draws a box within the window *win*, using the best possible characters that the terminal offers. If you want to draw a window around the whole window, you can use the curses function *box()* instead.

- `void wladdch (WINDOW *win, int ypos, int xpos, int type)`
  This function is rarely used by any application. It adds a single halfgraphical character to the specified position within the window *win*. The trick hereby is that a previously existing halfgraphical character is not overwritten, but instead, new branches will be added. Type is a set of the or'ed constants AI_UP, AI_DOWN, AI_LEFT and AI_RIGHT. If you already have a horizontal line and add AI_UP | AI_DOWN, you'll get a cross.

- `void whorline (WINDOW *win, int ypos)`
  Adds a horizontal line across the whole window at the vertical offset *ypos*. This line is drawn using the above function, and will therefore harmonize with other lines (or boxes).

- `void woutline (WINDOW *win, char *text, int ypos, int type)`
  Writes the text (which must be shorter than the horizontal size of the window) to the window at the vertical position *ypos*. It is written using the alignment specified by *type*, which may take the values AL_LEFT, AL_RIGHT, AL_CENTER or AL_BLOCK. With the exception of block alignment, the previous content of the window's line may not be completely overwritten (so that you can write both left-aligned and right-aligned text into one screen line).

- `void wprintstr (WINDOW *win, char *text, int ypos, int xpos)`
  Writes *text* without formatting to the position *ypos*/*xpos* inside of the window *win*. This may be some imaginary position. If *xpos*<0, then the beginning of the string is cut off. If text extends beyond the window's horizontal limits, it is cut off as well. This function should be able to handle all cases.

- `int InputText (WINDOW *win, char *result,`
  `int ypos, int xpos, int len, int type)`
  The text in *result* is printed to and the cursor is set to the specified position within window *win*, and the user may modify this data. The field's length is *len*. *type* is a set of or'ed flags. These flags are similar to those used in resource files for form field descriptions, which is hardly surprising since this function does all input for *QueryForm()*.

  There are the folllowing flags:

IT_ALPHA	allows input of alphabetic characters (upper- and lowercase)
IT_NUMERIC	allows input of digits
IT_ALNUM	combination of IT_ALPHA and IT_NUMERIC for alphanumeric chars
IT_UPCASE	transforms all lowercase characters to upper case upon input
IT_DCASE	transforms all uppercase characters to lower case upon input
IT_SPACE	allows the entering of the space character
IT_CHAR	allows the entering of other printable characters like '.' etc.
IT_TABEXIT	exits the function when the tab (or backtab) key is pressed
IT_FKEYEXIT	exits the function when a function key is pressed

  The function returns IT_OK, if the user confirms his input by pressing the return key. IT_ESCAPE, IT_BTAB, IT_TAB is returned if the user has pressed the escape, backtab or tabulator key, respectively. It will never return IT_BTAB or IT_TAB if the IT_TABEXIT flag is not set. If IT_FKEYEXIT is set, the function can also return the keycode for a function key (KEY_F(*n*)), where *n* is the number of the function key.

  Because the contents of *result* are written to the screen, it must contain a legal, printable character string (must not be of length *length*). The contents of this string may have changed even if the user has pressed the escape character. Upon exit, the string holds the changed data, eventually padded with spaces.

  Before calling this function, you should set the proper attributes for window *win*.

- `chtype int2acs (int type)`
  type is a set of the flags AI_UP, AI_DOWN, AI_LEFT and AI_RIGHT. This function will return a corresponding halfgraphical character, if supported by the terminal (and the curses library). int2acs (AI_UP | AI_LEFT) will produce a right lower corner. If *type* is an illegal combination (only AI_UP), the function returns a space.

- `int acs2int (chtype c)`
  The same, only vice-versa. If *c* is a halfgraphical character, this function returns the appropriate combination of the above mentioned flags. This does not work on systems where the curses library doesn't define halfgraphical characters.

The following two macros are usually defined within curses, however, not all implementations do define them, in which case they are defined by the curses library.

- `void getmaxyx (WINDOW *win, int y, int x)`
  Get the size of the window *win* and puts it into the variables *y* and *x*. This is a macro, and therefore does not need to be given the addresses of the variables, but the variables theirself.

- `void getbegyx (WINDOW *win, int y, int x)`
  Gets the absolute coordinates of the upper left corner of the window *win* and puts it into the variables *y* and *x*. This is a macro, and therefore does not need to be given the addresses of the variables, but the variables theirself.

Panel Functions

This chapter is intended for programmers on systems without an own panel library, who have to use *nopanel* and who don't have other documentation on the panel library calls. But since the *nopanel* functions behave exactly the same as the real functions, *panel* users might want to look at it as well.

Panels give a *window* depth. While a *window* cannot be above or below other windows, a *panel* can. The contents of a partially obscured *panel* do not get lost, they are merely covered by the other *panel* and get visible again once the above *panels* are removed. Terminology only gets a bit screwed up by the fact that a *panel* is not a complete screen data structure of its own but instead rides piggyback on a *window*. This underlying *panel* is responsible for the contents, while the additional *panel* information is responsible for screen update. Throughout this documentation, the term *association* was used to express the relationship between the *window* and its *panel*.

Generally, you should create a panel for each window. If windows are displayed on-screen without the panel's screen update functions, the screen will most likely get messed up. The only exception is the background window *stdscr*, which is correctly handled by the panel functions.

- `PANEL *new_panel (WINDOW *win)`
  Creates a new panel and associates it with the window *win*. The new panel is located on top of all other panels and is visible (this mans that the visibility flag is set, see below). The panel will appear on the screen when the screen is next updated; it will then reflect the contents of window *win*.

  This function only needs to be called once on each window; all past operations on the window will automatically be sent to the screen (that is, on each update).

  The function returns NULL on error.

- `int del_panel (PANEL *pan)`
  Deletes the panel *pan* from the screen. Note that only the panel is destroyed but not the window, which must be seperately destroyed with the curses *delwin()* function. The function returns ERR on error (that is, if *pan*=NULL); otherwise OK is returned. The panel stays visible until the next screen update.

- `int top_panel (PANEL *pan)`
  Moves the panel *pan* to the top of the panel stack so that it appears in front of all other panels. The function returns ERR if *pan*=NULL, otherwise OK. Don't forget to update the screen.

- `int bottom_panel (PANEL *pan)`
  Moves the panel *pan* to the bottom of the panel stack so that it appears behind all other panels. But it is still placed before the background window *stdscr*. The function returns ERR if *pan*=NULL, otherwise OK. Don't forget to update the screen.

- `void update_panels (void)`
  Synchronizes the contents of all panels with the contents of the screen; that means all changes are prepared to be sent to the screen. After this function the curses function *doupdate()* must still be called to actually send the data to the physical screen. Both functions are usually called together:

```
update_panels();
doupdate     ();
```

  Depending on the type of the screen an update may take its time (especially across networks or modem connections), thus you should call these functions as infrequent as possible. This helps curses to optimize the changes (like cursor movements etc). Usually it is sufficient to update the screen before keyboard input.

- `void hide_panel (PANEL *pan)`
  Clears the visibility bit of panel *pan*. This has the effect of making the panel invisible, and it will disappear from the screen on the next screen update. Hidden panels can still be handled as usual; they may be changed or moved etc. If the panel was already hidden before this function is called, the function has no effect.

- `void show_panel (PANEL *pan)`
  Restore a hidden panel at its previous depth, although newly created panels may be on top of it, and panels moved with *top_panel()* or *bottom_panel()* may also newly be on top of or below it.

- `WINDOW *panel_window (PANEL *pan)`
  Returns a pointer to the window that is associated to panel *pan*.

- `void replace_panel (PANEL *pan, WINDOW *win)`
  Replaces the window which is currently associated to the panel *pan* with *win*. The previously associated window is not deleted, instead, it is simply 'forgotten' by the panel. The programmer should therefore previously a pointer to it using *panel_window()* to delete it manually, or use it elsewhere.

- `PANEL *panel_above (PANEL *pan)`
  Returns a pointer to the panel which is above panel *pan*. If *pan* is on top of all other panels, NULL is returned. If this function is called with NULL as parameter, a pointer to the bottom panel is returned.

- `PANEL *panel_below (PANEL *pan)`
  Returns a pointer to the panel which is below panel *pan*. If pan is below all other panels (that is, the panel directly before *stdscr*), NULL is returned. If this function is called with NULL as parameter, a pointer to the top panel is returned.

- `void set_panel_userptr (PANEL *pan, char *ptr)`
  The user pointer is a pointer that is intended for use by the programmer only. It is never referenced by any other panel function (save *panel_userptr()*, of course). This function sets the user pointer of panel pan to ptr.

- `char *panel_userptr (PANEL *pan)`
  Returns the user pointer of panel *pan*. Before the user pointer is first set with a call to *set_panel_userptr()*, its value is undefined.

- `void move_panel (PANEL *pan, int ypos, int xpos)`
  Moves the panel and its associated window to the requested position. Never move a window itself, but only via this function. If you move a window and do not notify the panel, the screen might get messed up. Upon creation of a panel, the position of the window is automatically detected, therefore you do not need to call this function at that time.

- `int panel_hidden (PANEL *pan)`
  Returns true if the panel *pan* is hidden, or false if it is not.

The resource compiler

You can define resources in two ways. First, they can be defined in a resource file, then they can be included as internal resources. Especially during the initial test phase of a program, a resource file has certain advantages, because parameters can be changed without recompiling the application. But the resource file also has some disadvantages, because an external file must always be present. In some cases, a resource file must often be protected from the user, who could change a text file fairly easy.

The resource compiler was invented to compile a resource file into internal resources.

Compiling resources

The resource compiler reads the resource file and creates a C file. This C file declares one function that internally loads all resources that were declared in the resource file. This file may then be compiled as usually. Once it is linked to the application, and the loader function has been called, all resources are present.

The syntax of the resource compiler is as follows:

```
rescomp [-l <limit>] [-r] [-n <fname>] <rfile> ...
```

The program reads in the resource files by the name of `<rfile>` and all other filenames that were given on the command line. The resulting C program is written to the standard output. The loader function has the name `LoadInternalRes`; this name can be changed with the `-n` option.

The compiled C program uses static strings to store the resources. Only, several compilers limit the maximum size of static strings. Usually, strings up to 32kBytes are created; this limit is changed with the `-l` option. You can define the limit as bytes or scale it to kBytes with the 'k' suffix. The default value therefore could be generated with '`-l 32k`'.

If you give the program the '`-r`' option, it will also evaluate referenced resource file, which were included by the resource type 'Resource'. Evaluation means in this case that the contents of the referenced files are included. Without this option, these references are ignored, not even the reference itself is included.

The output of this program is a simple normal C file that must be compiled as any other module. The file <resource.h> is included, and you can give it the '-DNOPANEL' compiler option if you use *nopanel* instead of the default panel library. The resulting object file must then be linked to the application.

In your application, you can now replace the resource file's name in the *StartCurses()* call with NULL, and separately call the loader function of the internal resources:

```
if (StartCurses (NULL, NULL, monoflag) == FALSE) {
    printf ("Error!\n");
    exit   (99);
```

```
}
LoadInternalRes ();
```

Note that you are not limited to using only a single internal resource. Provided that you give different function names to the loader functions (the `-n` parameter), you can use as many internal resources as you want. Furthermore, you can mix resource files and internal resources.

Suggestions

You can quite simply arrange for building for internal and external resources simultaneously. The first step is to define a compilation rule for the resource file in the Makefile, e.g.:

```
programm.res.o :     programm.res
          rescomp -l 8k -n ProgrammRes programm.res > $(@F:.o=.c)
          $(CC) -c $(CFLAGS) $(@F:.o=.c)
```

This file must then as well be specified when linking, of course. Now, while starting up your application, you use the following code:

```
#ifndef USEINTERNAL
  if (StartCurses ("programm.res", NULL, monoflag) == FALSE)
    exit (99);
#endif
  if (StartCurses (NULL, NULL, monoflag) == FALSE)
    exit (99);
  ProgrammRes ();
#endif
```

Now, you can decide with a simple compiler flag if you want to use the internal or external resources. Of course, you can also push this approach one step further and implement a run-time command line flag; or you can use the external resource file if it is present, and load the internal resources if it is not.

But when the application leaves the development process, you should finally decide which resources should be external and which should be not. We should note again that resource files mean visibility of the resources to the user (which is frequently undesired), where internal resources mean that they are hidden from the user.

Here, we will propagate a mix. Of course a lot of resources, like the layout of forms, screen menus or message texts should not be subject to change at the user's will and therefore should be loaded internally. On the other hand, there are resources like the page width of a printer that might be frequently changed by the user. These parameters have "user level", that is, the user is assumed to have the ability to maintain them on his own.

The approach of mixing internal and external resources utilizes the mechanism that resources can be overwritten if they are loaded again. So you can set parameters with internal resources to useful values, and the user can change them by editing a resource file that is loaded after loading these internal resources. This resource file is then nothing but a configuration file, and it is common to load a configuration file by the name of '`.<app-name>rc`' from the user's home directory.

This of course does not prevent the user from redefining other 'hidden' internal resources, but since they are not documented to him, he'll have a difficult time to even guess their name. But if this

insecurity is still a concern to you, you can prevent this by loading 'hidden' resources from internal resources after the configuration file is loaded.

Besides a local configuration file from the user's home directory, a global (machine-wide) configuration file is usually loaded, so that a system administrator can set some default values for all users (machine-dependent parameters, like paths, printer names etc.). Utilizing all these mechanisms, the source code looks as simple as follows:

```
if (StartCurses (NULL, NULL, mono) == FALSE)
  exit (99);
LoadDefaults ();                  /* Load default values internally */
strcpy (fname, getenv ("HOME")); strcat (fname, "/.apprc");
AddMoreResources (LoadResoruce ("/usr/local/lib/apprc"));
AddMoreResources (LoadResource (fname));
LoadInternals ();                 /* load internal resources 'safely' */
```