TNG SOFT enterprises proudly presents

EasyVision 2.0

A text mode based user interface

The easy to use, reliable and powerful C and C++ library of functions, for the DOS environment.

This software is shareware, and must be registered.

TNG SOFT : The Next Generation Software

C H A P T E R 1 : Overview

Welcome to EasyVision!

This C and C++ library is a collection of routines for the DOS environment that were developed while working on different projects.

Because I am big on modularity, I always try to put general purpose code together in a separate function. Therefore, after developing an application, I am often left with many useful functions. Those can be recycled easily and reduce development time of other applications.

This package comes in two sections:

1. Day-to-day utility functions, written in C. They can be used in any program.

2. Object-oriented C++ classes, providing an easy to use, reliable and powerful text mode based user interface.

I've decided to make this code available to everyone through this library. All those functions were tested thoroughly, and put to work in actual applications.

Why EasyVision?

DOS? Even if the times are to WINDOWS programming, you may be obliged to go with a text mode application if the computers on which they're supposed to run are small machines. However, that doesn't mean that your implementation can't have a professional look!

Why reinvent the wheel when someone before you created one that works quite well?

One reason would be that professional programmers like to write all of their general purpose functions themselves. They want to know what's inside. But sometimes, you just don't have the time to do it, or simply the need for it.

This library will give you access to useful routines that will shorten your development time and make your code more reliable. It will effectively free you from having to write many of the common functions your programs might need.

EasyVision was first created as a need for a text mode based user interface. After looking at some shareware and commercial user interface packages, and reading comments about them from many C and C++ programmers, it was clear that something was missing.

Turbo Vision?

When people talked about TURBO VISION from BORLAND, it was their opinion that it was too difficult to use. In my own opinion, I think that TURBO VISION is one of the greatest work of software engineering around. It is the most powerful and professional text mode user interface in existence. It is so well implemented and thought out that it is the standard in text mode interfaces that everyone is following, including EasyVision! (Hope they don't sue me...) But, it is so big that it is a language in itself, and that's what makes people afraid of using it!

On the other hand, there are those shareware libraries. Some of them are extraordinarily well done, but are still much too big. I'm thinking about CXL right now. Others are too small and too unreliable to develop serious software.

So, what's a C++ programmer's to do? Maybe just what I did, and write all of his interface himself. But what a waste if I'm the only one using it! Why not make it available to everyone? Well, that's what EasyVision is all about!

What is EasyVision?

EasyVision is a collection of short C functions, dealing with the screen, the keyboard, text strings, etc.

It is also a text mode based, windowed user interface. It provides a DESKTOP, a STATUSLINE, a MENUBAR, WINDOWS, CONTEXT SENSITIVE ON-LINE HELP, MOUSE SUPPORT and much much more...

EasyVision was created with 2 important priorities:

1. It should be EASY to learn and EASY to use. Provide only the big, important functions to the user. Make sure that those functions are REALLY powerful and produce professional looking results!

   This library hasn't been written to provide every function needed to develop full featured word processors or the likes. It is there to give you a strong and reliable skeleton to build your programs on. It is up to you to come up with the 'meat'.

2. Those functions should be totally bug free and crash proof. EasyVision should validate all parameters to make sure nothing wrong can happen. Don't rely on the programmer's good will to check out its code for out of range or non-initialised parameters.

Demo programs

That was what EasyVision was supposed to be. Well, EasyVision is still better than that! At this point, if you haven't already done so, you should run the demonstration programs that are in the archives "DEMO1.ZIP" and "HANOI.ZIP", to see the results of not so many lines of C++ codes that uses the EasyVision library...

If you find the results interesting, it's up to you to go on. All functions and classes are FULLY documented in the following pages. The source codes of the demonstration programs are also included (and commented) in the archive. It provides you with 'real' examples of how to use this library.

Current Version

The complete history of EasyVision can be found in the "HISTORY.TXT" file, included in the archive.

A word about registration

EasyVision may be freely distributed, without charge except for the media cost.

EasyVision is made available under the shareware concept. This means that after an evaluation period of 30 days, you should register this software with its author. Furthermore, if you use this software to create your own shareware software, YOU MUST REGISTER EasyVision.

Registration grants you a life-time license to use this software, and all following versions or updates.

EasyVision is NOT crippled in any way. There is absolutely no difference between the registered or unregistered versions.

To register this software, complete the "REGISTER.TXT" registration form included in the archive. Registration is $25 CANADIAN. You will receive through 'snail mail' an official registered user certificate with your registration number. For 35$ CANADIAN, you'll also receive a bonded true type copy of the latest version of the manual.

EasyVision and TNG SOFT ENTERPRISES are registered trade marks.

What's Next?

Every time I write a general purpose function, and think it could be used in other project, I will include it in the EasyVision package. Every now and then, a new version of this library will be made public.

I will try to improve the functions already there. There won't be many new commands. EasyVision will always remain EASY to use, to leave the programmer at more important tasks. However, I welcome your suggestions to what you think is missing from this package.

I will not accept special demands. The purpose of this library is to distribute code that I have in my personal library.

About the Author

My name is Remy Gendron. I live in Quebec city, Canada. I'm studying at Laval University in computer sciences (Informatique de génie). I've been programming since I was 14 years old. I started on a Texas Instrument's TI-57 programmable calculator!

I then graduated to a Commodore 64. This machine was a breakthrough in computer power, ahead of its time. I've done some BASIC on it, but mostly assembler. Great machine to learn on. The C64 did cost me about as much as you would spend today on a 486! Those of you who really did work with this machine know what could be done with only 64K of memory, when one did put his mind on it (Remember GEOS?). I find it incredible that with the computer power we have today, we don't manage to do something better...

I was then away from computers for a couple of years. I returned with a real IBM (AT), then another Commodore (386sx-20), and finally a pieced together (386dx-40). I'm looking forward to a 486dx3-99 or maybe a Pentium?

User support

All of EasyVision's functions and classes have already been tested in real applications. They should behave as indicated. Please take time to carefully read the documentation. The answers to your questions should be in there. The demo programs should also provide a good introduction.

I will gladly answer any questions you may have. I'll also welcome any comments or suggestions. I can be reach by netmail/email on FIDONET or INTERNET:

FIDONET REMY GENDRON 1:240/1
INTERNET REMY_GENDRON@f1.n240.z1.fidonet.org

So, that's about it for now! Have fun and enjoy!

Remy Gendron
author of EasyVision

C H A P T E R 2 : Getting started

Installing and using the EasyVision library is very simple.

Library specifics

EasyVision's functions are for the DOS environment. They try, but do not always follow the ANSI standard.

This software was developed under BORLAND C++ 3.1.

➔ TURBO VISION's application framework was NOT used in any way to create EasyVision, nor any other 3rd party libraries. It was built from scratch over a period of two years.

Memory model — The code was compiled under the 'large' memory model. All prototypes were declared as 'far' functions and all pointers were explicitly declared 'huge'. This will provide full compatibility when linking to most memory model sizes. You should consult your compiler documentation about interfacing with different memory models.

If you should require that the library be compiled in a different memory model, just get in touch with me. We'll arrange something, if you're a registered user of course!

Pointers — Unless you REALLY know what you are doing, you should always use 'huge' pointers. 'far' pointers can cause wrap around and comparison problems because they're not normalized. All of EasyVision's functions use 'huge' pointers.

Video — The video output is done through direct screen writes. This makes for incredibly fast outputs. Going through the BIOS is just to slow. However, under multitaskers like DESKview, who often work in text mode, screen bleeds can occur if the application is running in the background. Use the virtualising options when running under DESKview.

Header files — All header files use conditional compilation to prevent redeclaration errors at compile time. So, if you're not sure if a header was previously included (possibly by another header file), feel free to include it again.

➔ They also provide for C++ compilation, using a conditional 'extern "C"' keyword.

Installation

Unpack the archive in a temporary directory. If you haven't done so, you should really print all the USER'S GUIDE for easier reading. It as been formatted to print at 60 lines per pages. And why not take a look at the header files.

Put all header files "*.H" and "*.HPP" into an INCLUDE directory. Just to be sure you won't overwrite existing header files, you should make a separate include directory, then include it in the INCLUDE path of your compilator.

Then put the EasyVision library "EVISION.LIB" into one of your LIBRARY directories.

How to use this library

To use a library function or class, just include its header file in your source code. YOU MUST NOT write a prototype yourself based on the prototypes written in this manual. The real prototypes may have additional information in them. So, for example:

```
#include <stdio.h>                                  /* A standard header file */
#include "stdfcts.h"                              /* An EasyVision header file */

void main (void)
{
  ...
                                          /* Here you can use the desired functions */
  ...
  return ;
}
```

You then have to link all your modules, including the "EVISION.LIB" library. You do that by including "EVISION.LIB" in your project. That's all there is to it!

➔ If you get errors, that's probably because you compiled your sources in C and included C++ modules. Another source of errors would be if you tried to linked different memory models. Remember the EasyVision was compiled under the 'large' memory model.

Validation All arguments to functions are FULLY validated. An EasyVision function will never let you get away if it is called incorrectly. If something is wrong, the program is stopped and a plain English error message tells you what went wrong, where and why! In the function descriptions, when it says that you SHOULD NOT or CANNOT do something, it means that if you do it, you'll get an error message. Your program will not crash!

How to use this document

The following conventions are used in this document:

➔ All of EasyVision's functions and methods were declared of type 'far'.

The following symbols are used in the text:

" Regular C and C++ keywords

{} EasyVision's keywords

<> Arguments to functions

➔ Important remarks (that you MUST read)

"" Filenames

CAPS Keyboard keys

Related functions are grouped together in the same module. A chapter is devoted to each module.

At the end, reference information can be found in appendixes.

C H A P T E R 3 : Using EasyVision's templates

EasyVision provides you with some starting templates for your programs. Templates are important facilities. They bring consistency and productivity in your programming style.

If you already have your own style and templates, stick with them. The included templates are for new programmers, or those of you who are still searching for a better way.

Of course, those are just suggestions. For instance, the usage and placement of parenthesis, currently generate many hot debates. Some will follow the 'professional' style and open a block this way:

```
while (condition) {
  statements ;
  statements ;
}
```

While others (and I), do it this way:

```
while (condition)
{
  statement ;
  statement ;
}
```

Choose your own style! The following files are therefore included for your convenience.

MAIN.C

Template for your main source file. This is the only file with a 'main' function declaration. I'd go further by suggesting that ONLY the 'main' function be included. As a rule, your 'main' should only call other functions in your other modules. It should be as short and as well documented as possible.

MAIN.CPP

C++ version of MAIN.C.

MODULE.C

Template for your secondary modules' code files. You could, for example, put all video related functions in a "VIDEO.C" file.

MODULE.CPP

C++ version of MODULE.C.

MODULE.H

Template for your secondary modules' header files.

MODULE.HPP

C++ version of MODULE.H.

STDMACRO.H

Template for your standard macro definitions. Put in this file your standard macros that can be used with many different projects.

➔ This file contains declarations for {TRUE} and {FALSE} and for all the keyboard keys (II_* macros).

PRJMACRO.H

Template for macros particular to your current project. Put in this file, macros that will be needed by many modules. You should place macros specific to a module in the module's macro declarations section.

STDTYPE.H

Template for your standard type definitions. Put in this file your standard typedefs that can be used with many different projects.

➔ This file contains declarations for the {bool}, {byte}, {word} and {dword} types.

PRJTYPE.H

Template for typedefs particular to your current project. Put in this file, typedefs that will be needed by many modules. You should place typedefs specific to a module in the module's typedef declarations section.

PRJMSGS.C

Template for your global output messages. When your program outputs something to the user, put it here as a global variable. Then, reference this variable everywhere in your program when you want to output this message. This will allow easy conversion of your program from one language to another.

I suggest that all global message variables begin with 'msg_'. For example, msg_fileselect.

PRJMSGS.H

Template to put 'extern' references to your global message variables. For example:

extern char huge *msg_fileselect.

HEADTEST.C

Template to test your header files. Often, they will compile correctly because some other files were included before your header. This could cause problems if you intend to use this header file elsewhere, or make them available to other programmers. Your header files should always compile alone by themselves. Test them with this file.

Examples

Templates promote consistency and free you from tedious tasks. Also, having something to start with, you won't as often be afflicted by the 'blank page' syndrome!

Take a look at the demo programs to get a feel at project management.

C H A P T E R 4 : Using EasyVision's functions

There are two types of functions in the EasyVision library. Functions available in the standard libraries, and new functions.

Because I like to compile in the 'large' memory model, all functions have been written to be of type 'far'. They will also accept 'huge' pointers without the need for a typecast. Those functions don't call the standard run-time library. They were entirely rewritten. There is no overhead and they are fully optimized.

New functions have been written because they weren't available in the standard libraries.

Conventions

➔ Some conventions have been adopted for the function names. Related functions will have the same name prefix. For instance, all string functions will begin by {str_}.

➔ Function declarations can use types like {bool}, {byte}, {word} and {dword}. See the file STDTYPE.H for a description of those types.

A function's description uses the following format:

Summary Short description of this function's behavior.

Syntax #include "header.h"

```
ReturnType FonctionName
(
  <param>,
  <param>
) ;
```

➔ YOU MUST NEVER WRITE A PROTOTYPE FOR A FUNCTION YOURSELF. ALWAYS USE THE PROPER HEADER FILES. THEY HAVE ADDITIONAL INFORMATION IN THEM!

Remarks Parameters and usage are described here when needed.

Return The returned value of the function is explained here.

Example Examples of various calls to this function.

C H A P T E R 5 : EasyVision's standard functions

The declarations for EasyVision's standard functions are contained in the "STDFCTS.H" header file.

Those are the truly general and often used functions that will be required by most of your modules.

ASSERT

Summary This function will ASSERTain that a <condition> is {TRUE}. If it is, it will return immediately with no effect, and minimum overhead.

If the condition is {FALSE}, the program will be terminated in an orderly fashion. {assert} will clear the screen, display an error message for at least five seconds, and terminate the program with a call to 'exit'. This closes all open files, releases any memory allocated on the heap and exits to DOS.

Syntax #include "stdfcts.h"

```
void far assert                                  /* Validates an assertion */
(
  bool           condition,                      /* FALSE=terminates */
  char      huge *fctname,                       /* Current function */
  const char huge *errortext,                    /* Error message */
  int            exitcode                        /* Errorlevel */
) ;
```

Remarks If <condition> evaluates to {FALSE}, the program will be terminated and <fctname> will be displayed. You should set <fctname> to the currently executing function. This will help find the location of the error.

The error message <errortext> will be displayed. This can be a string literal, or you can use predefined error messages. These are available from the global variable msg_stderr[]. This is an array of pointers, and here are the messages:

1. "Not enough memory to create an array on the heap."
2. "Not enough memory to create a struct on the heap."
3. "Not enough memory to allocate the requested amount of bytes."
4. "Out of memory."
5. "File not found."
6. "Path not found."
7. "File access denied."
8. "Input/Output error."
9. "Unrecoverable fatal error."

{assert} will then return to DOS with an errorlevel of <exitcode>.

➔ Before calling 'exit', {assert} will set the global variable {assert_err} to {TRUE}. This will allow classes' destructors to know the state of the program when they were called. At any other time, {assert_err} is {FALSE}.

Return None.

Example

```
assert (nbrecord>0,"datasearch","No data",1);
assert (ptr != NULL,"dataprocess",msg_stderr[3],1) ;
```

ARG_EXIST

Summary This function will check if an argument is present on the command line. Check is case sensitive.

Syntax #include "stdfcts.h"

```
int far arg_exist                                   /* Checks for argument */
(
  char huge *string                                 /* Argument to search for */
) ;
```

Remarks The search for the argument is case sensitive.

Return If the command line argument <string> exists, its index in the command line argument array will be returned. If the argument does not exist, the function will return 0. See the '_argv' keyword of your compiler for details on accessing command line arguments.

Example

```
if (arg_exist ("q")) sound = FALSE ;
```

ARG_IEXIST

Summary This function will check if an argument is present on the command line. Check is case insensitive.

Syntax #include "stdfcts.h"

```
int far arg_iexist                                  /* Checks argument */
(
  char huge *string                                 /* Argument to search for */
) ;
```

Remarks The search for the argument is case insensitive.

Return If the command line argument <string> exists, its index in the command line argument array will be returned. If the argument does not exist, the function will return 0. See the '_argv' keyword of your compiler for details on accessing command line arguments.

Example

```
if (arg_iexist ("bios")) directvideo = FALSE ;
```

HEAPALLOC

Summary {heapalloc} replaces 'farmalloc'.

Syntax #include "stdfcts.h"

```
void huge * far heapalloc                          /* Allocates far heap */
(
  dword nbytes                                     /* Asks for a block <nbytes> long */
) ;
```

Remarks Allocates <nbytes> bytes on the far heap. {dword} is an 'unsigned long int' and is declared in "STDTYPE.H".

Return This function returns a 'huge' pointer. Be sure the pointer variable, who will receive the pointer to the allocated memory, is of type 'huge' also.

Example

```
char huge *ptr ;
ptr = heapalloc (sizeof (object)) ;
```

HEAPFREE

Summary {heapfree} replaces 'farfree'.

Syntax #include "stdfcts.h"

```
void far heapfree                                  /* Deallocates heap */
(
  void huge *block                                 /* Ptr to block to free */
) ;
```

Remarks The only difference from 'farfree' is that this function offers the convenience of accepting a 'huge' pointer as an argument.

Return None.

Example

```
char huge *ptr ;
ptr = heapalloc (sizeof (object)) ;                    // Allocates memory
heapfree (ptr) ;                                       // Frees memory
ptr = NULL ;                                           // Always a good idea
```

TO_UPPER

Summary {to_upper} replaces 'toupper'.

Syntax #include "stdfcts.h"

```
int far to_upper                                       /* Converts to uppercase */
(
  int ch                                               /* Character to be converted */
) ;
```

Remarks The standard library 'toupper' function has a problem when you pass an 'int' to it. It considers only the LSB and could wrongly make the conversion.

Return An 'int'. The converted letter if it was necessary.

Example

```
command = to_upper (command) ;
```

TO_LOWER

Summary {to_lower} replaces 'tolower'.

Syntax #include "stdfcts.h"

```
int far to_lower                                       /* Converts to lowercase */
(
  int ch                                               /* Character to be converted */
) ;
```

Remarks The standard library 'tolower' function has a problem when you pass an 'int' to it. It considers only the LSB and could wrongly make the conversion.

Return An 'int'. The converted letter if it was necessary.

Example

```
command = to_lower (command) ;
```

C H A P T E R 6 : EasyVision's keyboard functions

The declarations for EasyVision's keyboard functions are contained in the "KEYBFCTS.H" header file.

GETKEY

Summary This is a replacement for the familiar 'getch' function.

Syntax #include "keybfcts.h"

```
int far getkey                              /* Reads a key from keyboard */
(
  int  filter,                                    /* Filter (0 = all keys) */
  bool buffer                               /* TRUE: Read from keyb buffer */
) ;
```

Remarks One of 'getch' weakness is its inability to deal with the way extended keys are internally represented. Those are the keys that don't have an ASCII code associated with them. For example, the function, arrow and editing keys all return an extended keycode.

This new {getkey} function will deal with these extended keys by adding 256 to the extended keycode. Appendix A lists all the extended keycodes currently available on an extended keyboard.

You MUST specify a <filter> to be used by the {getkey} function. A filter of 0 will allow any key to be read and returned by the function. You can also provide the ASCII or extended keycode (remember to add 256 to the real code) of the only key allowed to be returned by the function. This provides an easy way to WAIT FOR a specific key. The function will then return with that keycode, only when that key has been pressed.

➔ If you set <buffer> to {FALSE}, {getkey} will flush the keyboard buffer before reading a key. Otherwise, it will read from the buffer if keys have been previously pressed.

➔ Beware of NEVER setting <filter> to an impossible key entry, or you will be trapped by the {getkey} function!

Return If a normal key was pressed, the returned value is an 'int' representing the ASCII code of that key. (1-255)

If an extended key was pressed, the returned value is an 'int' representing the extended keycode plus 256. (256-396)

Example

```
getkey (0) ;
getkey (13) ;                                   /* Wait for the ENTER key */
```

EXTTOASCII

Summary Converts an ALT-(letter or digit) key combination to its equivalent, single letter or digit.

Syntax #include "keybfcts.h"

```
int far exttoascii                              /* Converts 'ALT-letter|digit' */
(
  int code                                      /* Code to be converted */
) ;
```

Remarks <code> is the value as returned by the {getkey} function. For example, {II_A_A} (Alt-A) is returned by {getkey} as 286. Given 286, {exttoascii} will then return 65, the ASCII code for the letter 'A'.

Return The ASCII code of the letter or digit in the ALT-key combination. If <code> is not an ALT-(letter or digit) combination, <code> is returned unchanged.

➔ If the value returned by this function is of a letter, the letter will always be promoted to uppercase.

Example

```
in = getkey (0,FALSE) ;
in = exttoascii (in) ;                          /* Checks for Alt-key combi */
```

C H A P T E R 7 : EasyVision's file functions

Example The declarations for EasyVision's file functions are contained in the "FILEFCTS.H" header file.

FNEWLINE

Summary Goes to start of next line in a text file.

Syntax #include "filefcts.h"

```
int far fnewline                                    /* Goes to next line */
(
  FILE *f                                           /* Ptr to opened stream */
) ;
```

Remarks The end of a text line is marked by the character '\n'.

Return If the start of a new line was found, '\n' is returned. If 'EOF' (end of file) was reach before a new line, 'EOF' is returned.

Example

```
error = fnewline (text) ;
```

FSIZE

Summary Returns size of a file in bytes, as displayed from a directory listing.

Syntax #include "filefcts.h"

```
dword far fsize                                     /* Returns file size in bytes */
(
  FILE *f                                           /* Ptr to opened stream */
) ;
```

Remarks In a text file, the new line character is in fact two bytes long. DOS stores '\n' as a combination of '\n' and '\r'. So, the length of a text file will not match the number of characters read with 'fgetc'.

Return {fsize} returns the size of the file in bytes as a {dword}. {dword} is defined in "STDTYPE.H" as an 'unsigned long int'.

Example

```
size = fsize (text) ;
```

FCOPY

Summary Copies a file.

Syntax #include "filefcts.h"

```
int far fcopy                                   /* Copies a disk file */
(
  char huge *srcpath,                           /* Path to source file */
  char huge *destpath,                          /* Path to destination file */
  bool     verify                               /* TRUE: Verify ON */
) ;
```

Remarks <srcpath> is copied to <destpath>. Paths are expressed in the form [d:][path]filename.ext.

➔ No wildcards are allowed.

If <verify> is set to {TRUE}, the two files will then be reread and compared. This is different from DOS's 'verify', who doesn't reread the files, and is therefore a little slower.

Return If the copy was successful and error free, {fcopy} returns 0. On error, 1 is returned.

Example

```
error =fcopy ("autoexec.bat","autoexec.bak",TRUE) ;
```

C H A P T E R 8 : EasyVision's screen functions

The declarations for EasyVision's screen functions are contained in the "SCRFCTS.H" header file.

SCR_TEXTATTR

Summary This function will set the background and foreground colors used for video outputs by the "CONIO.H" module. It will support bright background colors.

Syntax #include "scrfcts.h"

```
void far scr_textattr                          /* Sets colors of outputs */
(
  int back,                                    /* Background color */
  int fore                                     /* Foreground color */
) ;
```

Remarks The normal 'textbackground' function does not allow bright background colors. {scr_textattr} will do so.

➔ {tdesktop::settextmode] allows you to turn on the ability to use bright background colors. When this is so, all sixteen possible colors can be used when you have to specify a background color anywhere in EasyVision. Therefore, all changes of colors should be done with calls to {scr_textattr}.

You will now be able to have 'BLACK' text on a (really) 'WHITE' background.

Return None.

Example

```
scr_textattr (WHITE,BLACK) ;
```

SCR_VSAVE

Summary This function will save the current screen and video attributes in a 'text_info' structure. It has exactly the same effects as a call to 'gettextinfo'. The difference is that it can also save the actual screen content.

Syntax #include "scrfcts.h"

```
void far scr_vsave                             /* Saves all video informations */
(
  struct text_info huge *ti,                   /* Ptr to text_info */
  char huge*huge *savedscr                     /* Ptr to ptr */
) ;
```

Remarks All members of the 'text_info' structure <ti> are filled with this function. <savedscr> is a huge pointer to a huge char pointer. If this argument is provided, the screen content will also be saved to a buffer in the heap, and the huge pointer to char will be set to point to this buffer. If you don't want to save the screen area, set <savedscr> to 'NULL'.

➔ The cursor's attributes are saved, but will not be restored by {scr_vrestore}. You must use the {scr_csave} and {scr_crestore}.

Return None.

Example

```
struct text_info ti ;
char huge *scrbfr ;
scr_vsave (&ti,&scrbfr) ;
```

SCR_VRESTORE

Summary This function will restore the screen video mode and the text window size from a 'text_info' structure. It can also restore the screen content if it was saved by a previous call to {scr_vsave}.

Syntax #include "scrfcts.h"

```
void far scr_vrestore                          /* Restores video info */
(
  struct text_info huge *ti,                   /* Ptr to text_info */
  char huge*huge *savedscr                     /* Ptr to ptr */
) ;
```

Remarks The 'text_info' structure must have been previously filled with {scr_vsave}. The same is true for the previous screen content.

➔ If the screen area was not previously saved, set <savedscr> to 'NULL'.

The memory will be deallocated when the screen content is restored, so you can't restore it more than once.

Return None.

Example

```
scr_vrestore (&ti,&scrbfr) ;
```

SCR_CSAVE

Summary This function will save all cursor attributes, INCLUDING THE CURSOR TYPE (shape), in a {cur_info} structure. Those attributes are: colors, x pos, y pos and cursor shape.

Syntax #include "scrfcts.h"

```
void far scr_csave                          /* Saves all cursor attributes */
(
  struct cur_info huge *ci                  /* Ptr to cur_info */
) ;
```

Remarks The {cur_info} structure is as follows, and is defined in "SCRFCTS.H".

```
struct cur_info
{
  byte attribute ;                          /* Cursor colors */
  byte curx ;                               /* Cursor X position */
  byte cury ;                               /* Cursor Y position */
  word curtype ;                            /* Cursor type */
} ;
```

Return None.

Example

```
struct cur_info ci ;
scr_csave (&ci) ;
```

SCR_CRESTORE

Summary This function will restore all cursor attributes, INCLUDING THE CURSOR TYPE (shape), from a {cur_info} structure.

Syntax #include "scrfcts.h"

```
void far scr_crestore                       /* Restores cursor attributes */
(
  struct cur_info huge *ci                  /* Ptr to cur_info */
) ;
```

Remarks The cursor's attributes must have been previously saved with {scr_csave}.

Return None.

Example

```
struct cur_info ci ;
scr_csave (&ci) ;
scr_crestore (&ci) ;
```

C H A P T E R 9 : EasyVision's string functions

The declarations for EasyVision's string functions are contained in the "STRFCTS.H" header file.

STR_LEN

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
size_t far str_len                          /* Returns the length of a string */
(
  char huge *string                         /* Huge ptr to a string */
) ;
```

Remarks The only difference with 'strlen' is that this function offers the convenience of accepting a 'huge' pointer as argument.

Return The length of the string. The type 'size_t' is defined in "STDIO.H" as an 'unsigned int'.

Example

```
length = str_len (text) ;
```

STR_CPY

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
char huge * far str_cpy                     /* Copies a string */
(
  char huge *dest,                          /* Destination array */
  char huge *src                            /* Source string */
) ;
```

Remarks The only difference from 'strcpy' is that this function offers the convenience of accepting 'huge' pointers as arguments.

Return A 'huge' pointer to the destination string.

Example

```
char huge *string ;
string = heapalloc (20) ;
str_cpy (string,"Hello there !") ;
```

STR_CMP

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
int far str_cmp                          /* Compares one string to another */
(
  char huge *string1,                                    /* First string */
  char huge *string2                                    /* Second string */
) ;
```

Remarks The only difference from 'strcmp' is that this function offers the convenience of accepting 'huge' pointers as arguments.

Return {str_cmp} returns a value that is :

< 0, if <string1> is less than <string2>
= 0, if <string1> is the same as <string2>
> 0, if <string1> is greater than <string2>

Example

```
if (str_cmp (name[i],name[i+1]) > 0)
{
  /* Invert string (bubble sort) */
}
```

STR_ICMP

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
int far str_icmp                        /* Case insensitive compares */
(
  char huge *string1,                                    /* First string */
  char huge *string2                                    /* Second string */
) ;
```

Remarks The only difference from 'stricmp' is that this function offers the convenience of accepting 'huge' pointers as arguments.

Return {str_icmp} returns a value that is :

< 0, if <string1> is less than <string2>
= 0, if <string1> is the same as <string2>
> 0, if <string1> is greater than <string2>

Example

```
if (str_icmp (name[i],name[i+1]) > 0)
{
  /* Invert string (bubble sort) */
}
```

STR_TOUPPER

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
void far str_toupper                    /* Sets string to upper case */
(
  char huge *string                     /* Huge ptr to a string */
) ;
```

Remarks None.

Return None.

Example

```
str_toupper (name) ;
```

STR_TOLOWER

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
void far str_tolower                    /* Sets string to lower case */
(
  char huge *string                     /* Huge ptr to a string */
) ;
```

Remarks None.

Return None.

Example

```
str_tolower (name) ;
```

STR_PASTOC

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
void far str_pastoc                          /* Converts a string to c */
(
  char huge *string                          /* Ptr to string */
) ;
```

Remarks All characters in the string will be shifted left 1 space and a '\0' will be appended at the end of the string.

This is useful is you're reading records written in PASCAL format.

Return None.

Example

```
str_pastoc (name) ;
```

STR_CTOPAS

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
void far str_ctopas                          /* Converts a string to pascal */
(
  char huge *string                          /* Ptr to string */
) ;
```

Remarks All characters in the string will be shifted right 1 space, overwriting the terminating '\0'. The length of the string will then be put in the first byte of the array.

➔ The string MUST be 255 or fewer characters long.

This is useful is you're reading records written in PASCAL'S format, converted them to C'S with {str_pastoc}, then reconverting to PASCAL's before writing to disk.

Return None.

Example

```
str_ctopas (name) ;
```

STR_TRIM

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
void far str_trim                                   /* Normalises string */
(
  char huge *string                                 /* Ptr to string */
) ;
```

Remarks Useful when normalising an input.

Return None.

Example

```
str_trim (name) ;
```

STR_INVNAMES

Erreur! Entrée de glossaire non définie. ***Summary*** This

Syntax #include "strfcts.h"

```
void far str_invnames                               /* Inverts string */
(
  char huge *string                                 /* Ptr to string */
) ;
```

Remarks A name string is composed of 1 or more clusters of characters. Clusters are packets separated from each other by 1 or more spaces.

{str_invnames} will take the first cluster and move it to the end of the string.

This could be useful if you're doing a sort by last names then first names, but your name strings are in the form first then last.

Return None.

Example

```
str_invnames (name) ;
"Remy"              becomes "Remy"
"Remy Gendron"      becomes "Gendron Remy"
"Remy J. Gendron"   becomes "J. Gendron Remy"
"This is some string" becomes "is some string This"
```

STR_CENTER

Summary This function centers a string within a given area.

Syntax #include "strfcts.h"

```
char huge * far str_center                          /* Centers a string */
(
  char huge *string,                                  /* Ptr to string */
  int      area_length                               /* Length of area */
) ;
```

Remarks The original <string> is not modified. The new string will be padded with blanks, if necessary, to be centered in <area_length>.

The function will discard the part of the string that is longer than 255 characters.

Return This function will create a new string, and return a ptr to a static string defined in the string module.

➔ You MUST not try to free this string.

Example

```
printf (str_center ("Center of the screen",80)) ;
```

C H A P T E R 10 : EasyVision's time functions

The declarations for EasyVision's time functions are contained in the "TIMEFCTS.H" header file.

TICKTIMER_INSTALL

Summary This function installs or removes a tick counter.

Syntax #include "timefcts.h"

void far ticktimer_install (void) ; /* Installs timer */

Remarks Ticks are PC's clock units. They are 1/18 second long. The ticktimer routines will allow you to count those ticks.

The first call to {ticktimer_install} will install an interrupt, whose purpose is to count ticks. A second call to the {ticktimer_install} will remove the interrupt.

Return None.

Example

ticktimer_install () ;

TICKTIMER_RESET

Summary This function resets the tick count to zero.

Syntax #include "timefcts.h"

void far ticktimer_reset (void) ; /* Resets counter */

Remarks After installation of the interrupt, the tick count is undefined. You must reset the count before using the {ticktimer_read} function. Also, you can reset the count to zero anytime you like.

➔ Using this function will introduce a random delay of up to 1/18 second. The function will wait for the beginning of the next tick before returning. This will warranty that tick #1 will really be 1 tick long.

Return None.

Example

ticktimer_reset () ;

TICKTIMER_READ

Summary This function returns the current tick count.

Syntax #include "timefcts.h"

```
dword far ticktimer_read (void) ;                    /* Returns count */
```

Remarks After installation of the interrupt, the tick count is undefined. You must reset the count before using the {ticktimer_read} function. Using this function does not reset the counter.

Return The number of ticks since {ticktimer_reset} was last used. {ticktimer_read} returns a {dword}. {dword} is defined in "STDTYPE.H" as an 'unsigned long int'.

Example

```
dword count ;
ticktimer_install () ;                               /* Installs timer */
ticktimer_reset () ;                                 /* Resets timer to 0 */
function (argument) ;                                /* Calls your function */
count = ticktimer_read () ;                          /* Gets tick count */
ticktimer_install () ;                               /* De-installs timer */
```

DIFFDATE

Summary Calculates absolute number of days between two dates.

Syntax #include "timefcts.h"

```
dword far diffdate                                   /* Number of days between dates */
(
  int day1,                                          /* 1st date: day  (1-31) */
  int month1,                                        /* 1st date: month (1-12) */
  int year1,                                         /* 1st date: year (xxxx) */
  int day2,                                          /* 2nd date: day  (1-31) */
  int month2,                                        /* 2nd date: month (1-12) */
  int year2                                          /* 2nd date: year (xxxx) */
) ;
```

Remarks day1/day2 : Days of date 1 and 2. (1-31)
month1/month2 : Months of date 1 and 2. (1-12)
year1/year2 : Years of date 1 and 2. (1583 and after)

Return The absolute number of days between the 2 dates. {diffdate} returns a {dword}. {dword} is defined in "STDTYPE.H" as an 'unsigned long int'.

Example

```
dword count ;
count = diffdate (userday, usermonth, useryear,
        todayday, todaymonth, todayyear) ;
```

C H A P T E R 11 : EasyVision's miscellaneous functions

The declarations for EasyVision's miscellaneous functions are contained in the "MISCFCTS.H" header file.

ANSICOLOR

Summary Converts normal background and foreground color codes to color codes for ANSI escape sequences.

Syntax #include "miscfcts.h"

```
void far ansicolor                          /* Converts color codes to ANSI */
(
  int     fore,                             /* Fore and back colors */
  int     back,
  int huge *ansbold,                        /* ESC sequence */
  int huge *ansfore,
  int huge *ansback
) ;
```

Remarks fore, back : Standard screen colors.
*ansbold : Ptr to int to return bold value.
*ansfore : Ptr to int to return fore value.
*ansback : Ptr to int to return back value.

To interpret ANSI escape sequences, the ANSI.SYS driver (or its equivalent) must be loaded.

Return None.

Example

```
int bold, fore, back ;
ansicolor (LIGHTGREEN,BLACK,&bold,&fore,&back) ;
printf ("\x1b[%d;%d;%dm",bold,fore,back) ;
```

C H A P T E R 11 : Using EasyVision's classes

The EasyVision library has five classes. They provide a desktop, a statusline, a menubar, windows, user inputs and mouse support. Those classes are fully described in the following pages.

➔ All classes interact with one another. Don't try to use only one of them. You must work with EasyVision as it was meant to be used.

Every method is declared as 'far' and when they take pointers as parameters, those pointers are of type 'huge'.

Conventions

Conventions have been adopted for the methods' names. Related methods will have the same name prefix. For example, all window related functions begin by {win_}.

➔ Methods declarations can use types like {bool}, {byte}, {word} and {dword}. See the file "STDTYPE.H" for a description of those types.

➔ Parameters to C++ functions can have default values. If you don't supply a value, the default value will be used. However, rules apply to this usage of default parameters. Refer to your C++ documentations for an explanation of those rules.

A classe's description uses the following format:

First, the description of the class itself, its behavior, how it is related to the other classes and the interface. Then, each member function of the class is presented as in the following:

Summary Short description of this function's behavior.

Syntax #include "header.h"

```
ReturnType FunctionName
(
  <param>,
  <param>
) ;
```

➔ YOU MUST NEVER WRITE YOURSELF A PROTOTYPE FOR A FUNCTION. ALWAYS USE THE PROPER HEADER FILES. THEY HAVE ADDITIONAL INFORMATION IN THEM!

➔ C++ functions' header provide for optional arguments. IF YOU WANT TO INCLUDE AN OPTIONAL PARAMETER, ALL PARAMETERS BEFORE THAT ONE MUST BE INCLUDED AS WELL. Refer to your compiler's documentation.

Remarks Parameters and usage are described here when needed.

Return The returned value of the method is explained here.

Example Examples of various calls to this method.

Using classes is quite easy!

Windows For the {twindow} class, you firsts declare a pointer to this class. Then you instantiate (create) an object of this type with the operator 'new'. Now you can call the classe's member functions with the '->' operator. When you're finished, you free the memory taken by this class instance with 'delete'.

The others For the other classes, {tdesktop}, {tstatusline}, {tmenubar} and {tinput}, you can only have one instance of each. To make it easier, they all have been instantiated. You don't need to create them. You just use them right away. You access their member functions with the '.' operator.

Many examples are available in the source codes of the EasyVision's demo programs.

C H A P T E R 12 : EasyVision's tdesktop class

The {tdesktop} class is the first one to be used in a program that uses the EasyVision user interface. This class is responsible for the background on which the statusline, the menubar and windows will be displayed.

This class will save the screen content before the program was executed, initialise the video screen to the video mode of your choice, and display the desktop. You will also call this class at the end of your program to restore the previous video mode, restore the previous screen and reset default colors and cursor position.

➔ An instance has already been globally declared and is called desktop. Only one instance of {tdesktop} can be used in a program. As a result, you will never declare and create another instance.

➔ The desktop allows your programs to use bright background colors.

The desktop has built in default values. Only one call is required to do the work. The desktop will autosize itself to the screen. However, if you would like to change those default behaviors, other functions are provided to do so.

On the following pages, you will find each of its member functions.

Examples of using the desktop object are given in the source codes of the EasyVision's demo programs.

TDESKTOP::SETTEXTMODE

Summary Sets the textmode in which {open} will draw the desktop. Also sets the availability of bright background colors.

Syntax #include "tdesktop.hpp"

```
void far tdesktop::settextmode                          // Sets textmode
(
  int  mode=C80,                                    // Textmode code number
  bool brightbackground=FALSE                         // True = bright colors
) ;
```

Remarks This has no effect on the current textmode. It will only take effect when {open} is called. This call is optional. If it is not made before a call to {open}, textmode C80 (3, color 80 columns) is assumed and bright background colors will not be used.

Symbolic	Value	Text mode
LASTMODE	-1	Previous text mode
BW40	0	Black and white, 40 cols
C40	1	Color, 40 columns
BW80	2	Black and white, 80 cols
C80	3	Color, 80 columns
MONO	7	Monochrome, 80 columns
C4350	64	EGA 43-line, VGA 50-line

To use the symbolic constants, "CONIO.H" must be included.

➔ By setting <brightbackground> to {TRUE}, you will then be able to use all 16 colors for the background. As a drawback, blinking characters won't be available.

Return None.

Example

```
desktop.settextmode (C4350,TRUE) ;
```

TDESKTOP::GETSIZE

Summary Returns desktop's current size.

Syntax #include "tdesktop.hpp"

```
void far tdesktop::getsize                          // Returns desktop's size
(
  int huge *height,                                 // Ptrs to ints to return size
  int huge *width
) ;
```

Remarks The size returned in <height> and <width> is the ACTUAL desktop's size. If you changed the video mode with {settextmode}, but haven't called {open} yet, the new size won't be returned.

➔ When other parts of your program need to know the actual desktop's size, they should use this function instead of 'gettextinfo'.

Return None.

Example

```
int screenheight ;
int screenwidth ;
desktop.getsize (&screenheight,&screenwidth) ;
```

TDESKTOP::SETDESKCOLORS

Summary Sets the colors in which {open} or {refresh} will draw the desktop. Also selects the clock's colors.

Syntax #include "tdesktop.hpp"

```
void far tdesktop::setdeskcolors                    // Desktop's colors
(
  int back=LIGHTGRAY,                    // Desktop's background color
  int fore=BLUE,                          // Desktop's foreground color
  int clockback=RED,                                    // Clock's color
  int clockfore=WHITE
) ;
```

Remarks This has no effect on the current colors. It will only be used when {open} or {refresh} will be called. This call is optional. If {setdeskcolors} is not called before a call to {open}, 'BLUE' on 'LIGHTGRAY' is assumed for the desktop and 'WHITE' on 'RED' for the clock. See appendix B for a list of available colors, color codes and symbolic constants.

➔ The desktop displays an interrupt driven clock in the upper right corner.

Return None.

Example

```
desktop.setdeskcolors (BLACK,LIGHTGRAY,BLUE,WHITE) ;
```

TDESKTOP::SETTEXTURE

Summary Sets the character with which {open} or {refresh} will draw the desktop.

Syntax #include "tdesktop.hpp"

```
void far tdesktop::settexture                    // Set's desktop texture
(
  char asciicode=#176                                 // ASCII code to use
) ;
```

Remarks You must provide the character used to draw the desktop, in the form of its ASCII code. Only ASCII codes greater or equal to 32 are accepted. This has no effect on the current screen. It will only be used when {open} or {refresh} will be called. This call is optional. If {settexture} is not called before a call to {open}, ASCII code 176 is assumed.

Return None.

Example

```
desktop.settexture (' ') ;                                    // Plain desktop
```

TDESKTOP::SETTITLE

Summary Sets the title, and title colors displayed at the very top line of the desktop.

Syntax #include "tdesktop.hpp"

```
void far tdesktop::settitle                                   // Sets desktop's title
(
  char huge *text=NULL,                                       // Desktop's title
  int  back=EV_DEF,                                           // Title's background color
  int  fore=EV_DEF                                            // Title's foreground color
) ;
```

Remarks The title's colors are optional. If they are not provided, the desktop's colors will be used. This is used only if you do not intend to have a menubar, or if you will have something displayed before the menubar is created. This has no effect on the current desktop. It will only be used when {open} or {refresh} will be called.

This call is optional. If {settitle} is not called before a call to {open}, no title will be displayed. To reset a previously defined title, don't use any argument. You can give a <text> pointer argument that points to a text of any length, but only the part of the title that will fit on the titlebar will be displayed. So don't worry about the length of your title...

Return None.

Example

```
desktop.settitle ("EasyVision 2.0",RED,BLACK) ;
```

TDESKTOP::OPEN

Summary Opens the desktop.

Syntax #include "tdesktop.hpp"

```
void far tdesktop::open () ;                                  // Opens desktop
```

Remarks This will save the screen before the program was started, initialise the video screen and then display the desktop according to the default values, or those set by the previous functions. This call is NOT optional.

➔ You cannot reopen an already opened desktop. You must use the {refresh} function for that action, or first close it.

If the desktop as been closed, it can then be re-opened. This could be done for instance if you were to shell to DOS or to another program.

Return None.

Example

```
desktop.open () ;
```

TDESKTOP::CLOSE

Summary Closes the desktop.

Syntax #include "tdesktop.hpp"

```
void far tdesktop::close () ;                    // Restores screen
```

Remarks This will restore the video screen as it was before the desktop was opened. This call is NOT optional. You must first {close} the desktop with this function if you want to re-open it.

➔ {close} does not reset any of the desktop settings. You can easily re-open it after a shell to DOS for example.

Return None.

Example

```
desktop.close () ;
```

TDESKTOP::REFRESH

Summary The desktop has a {refresh} function that will redraw the screen. Using this function can be DANGEROUS...

Syntax #include "tdesktop.hpp"

```
void far tdesktop::refresh () ;                  // Redraws desktop
```

Remarks The refresh function will use the current values for colors, texture, etc... not the values when it was opened. This means that you can change the desktop colors for instance, and then {refresh} it.

➔ IN EASYVISION, WINDOWS CANNOT BE REFRESHED. YOU MUST MAKE ABSOLUTELY CERTAIN THAT THIS FUNCTION IS NOT AND CANNOT BE CALLED WHEN WINDOWS ARE OPENED, OR YOU WILL BE IN BIG TROUBLE.

Return None.

Example

```
desktop.refresh () ;
```

C H A P T E R 13 : EasyVision's tstatusline class

The {tstatusline} class allows your program to easily display information on the last line of the screen.

➔ An instance has already been globally declared and is called statusline. Only one instance of {tstatusline} can be used in a program. As a result, you will never declare and create another instance.

The statusline has built in default values. Only one call is required to do the work. The statusline will autosize itself to the screen. However, if you would like to change those default behaviors, other functions are provided to do so.

You can manually use the statusline by calling its {display} member function. Most of the time however, you won't. Each time you create an object in EasyVision, let it be a window, a button or an input field, you assign a short text to it. When this object is selected, this text will be automatically displayed on the statusline.

On the following pages, you will find each of its member functions.

Examples of using the statusline object are given in the source codes of the EasyVision's demo programs.

TSTATUSLINE::SETCOLORS

Summary Sets the background, foreground and highlight colors used by the {display} member function, when writing to the statusline.

Syntax #include "tstatusline.hpp"

```
void far tstatusline::setcolors                    // Sets colors
(
  int back=LIGHTGRAY,                              // Background color
  int fore=BLACK,                                  // Foreground color
  int high=RED                                     // Highlight color
) ;
```

Remarks All arguments are optional. If they are not provided, <back> defaults to 'LIGHTGRAY', <fore> to 'BLACK' and <high> to 'RED'. You can use color macros if "CONIO.H" is included. Appendix B gives a description of available color codes and macros.

The statusline is not modified by a call to this function. The changes to the colors will be made at the next call to {display}.

Return None.

Example

```
statusline.setcolors (RED,BLACK,YELLOW) ;
```

TSTATUSLINE::DISPLAY

Summary Displays a message on the statusline.

Syntax #include "tstatusline.hpp"

```
void far tstatusline::display                          // Displays message
(
  char huge *text=NULL                                 // Text to be displayed
) ;
```

Remarks The <text> argument can point to a message of any length, including the '~' characters. The message will be truncated to fit on the message area, which length is 68 characters. There is no need to clear the statusline before using this function. You can toggle between normal foreground statusline color and the highlight color with the special character '~' (tilde). To clear the statusline, call this function with no argument.

Return None.

Example

```
statusline.display ("Welcome to ~EasyVision~") ;
statusline.display () ;                                // Clears statusline
```

TSTATUSLINE::GETMSG

Summary Returns last displayed message.

Syntax #include "tstatusline.hpp"

```
char huge* far tstatusline::getmsg () ;                // Last message
```

Remarks Lets say you have a function that will modify the statusline. You could save its current content before modifying it, then restore it at the function's end.

Return None.

Example

```
char lastmsg ;
lastmsg = statusline.getmsg () ;
```

TSTATUSLINE::REFRESH

Summary The statusline object has a {refresh} function that will redraw it on the screen.

Syntax #include "tstatusline.hpp"

void far tstatusline::refresh () ; // **Redraws**

Remarks {refresh} will redisplay the last message.

➔ The refresh function will use the current color values. This means that you can change the statusline's colors and then, {refresh} it with the new colors.

Return None.

Example

```
statusline.refresh () ;
```

C H A P T E R 14 : EasyVision's tinput class

The {tinput} class is designed to process input information. It provides mouse support and gives your programs context sensitive help.

➔ An instance has already been globally declared and is called input. Only one instance of {tinput} can be used in a program. As a result, you will never declare and create another instance.

The {tinput} class has built in default values. However, if you would like to change those default behaviors, you can easily do so.

On the following pages, you will find each of {tinput}'s member functions.

Examples of using the input object are given in the source codes of the EasyVision's demo programs.

TINPUT::MOUSE_INIT

Summary This function will look for a mouse driver and initialise it. Initially, the mouse will be hidden.

Syntax #include "tinput.hpp" ;

```
void far tinput::mouse_init () ;                    // Initialises mouse
```

Remarks This function is called as part of the initialisation routines, before your program starts. You should never have to call it yourself.

➔ Remember that the mouse's cursor should ALWAYS be off when you make changes to the screen. Many older driver would leave garbage on the screen if you did updates while the cursor is ON. Anyway, the only time the mouse's cursor should be on is when you do user inputs. The {get} function will take care of all this and if you're doing things right, your only inputs will come from {get}.

If no mouse is found, calling the other mouse functions won't produce any result.

Return None.

Example

```
input.mouse_init () ;
```

TINPUT::MOUSE_STATUS

Summary Returns current mouse's state.

Syntax #include "tinput.hpp" ;

int far tinput::mouse_status () ; // **Mouse's status**

Remarks None.

Return Tree values can be returned:
0: No mouse's driver or mouse was found.
1: A mouse is present and the cursor is on the screen.
2: A mouse is present and the cursor is hidden.

Example

```
if (input.mouse_status () == 0)
  printf ("A mouse is necessary for this program") ;
```

TINPUT::MOUSE_SHOW

Summary Turns the mouse's cursor on.

Syntax #include "tinput.hpp" ;

void far tinput::mouse_show () ; // **Shows cursor**

Remarks If a mouse wasn't found, this function doesn't have any effect.

Return None.

Example

```
input.mouse_show () ;
```

TINPUT::MOUSE_HIDE

Summary Turns the mouse's cursor off.

Syntax #include "tinput.hpp"

void far tinput::mouse_hide () ; // **Hides cursor**

Remarks If a mouse wasn't found, this function doesn't have any effect.

Return None.

Example

```
input.mouse_hide () ;
```

TINPUT::MOUSE_LB_DOWN

Summary Tells if mouse's left button is currently pressed.

Syntax #include "tinput.hpp" ;

```
bool far tinput::mouse_lb_down () ;                    // Tells if down
```

Remarks None.

Return {TRUE} if mouse's left button is currently down. {FALSE} otherwise.

Example

```
if (input.mouse_lb_down ())
  do_something () ;
```

TINPUT::MOUSE_POS

Summary Returns mouse's screen position.

Syntax #include "tinput.hpp" ;

```
void far tinput::mouse_pos                          // Gets mouse's position
(
  int huge *row,                                    // To return row position
  int huge *col                                      // To return col position
) ;
```

Remarks If a mouse wasn't found, the returned values are not changed.

Return <row> returns the cursor's row position, 1 being the top row. <col> returns the cursor's column position, 1 being the leftmost column.

Example

```
int row, col ;
input.mouse_pos (&row,&col) ;
```

TINPUT::GET

Summary Gets a user input, be it a keyboard or a mouse event. Provides automatic context sensitive help and mouse support during input.

Syntax #include "tinput.hpp" ;

```
int far tinput::get                                  // Gets a keyb or mouse input
(
  input_info huge *ii=NULL,                          // Ptr to input_info
  int          filter=0,                             // Valid key codes
  int          hlpctx=EV_NOHLPCTX,                   // Context
  bool          buffer=FALSE                         // Use buffer?
) ;
```

Remarks This function is at the heart of EasyVision's input facilities.

➔ One important concept is that an EasyVision's input is ALWAYS stored in an {input_info} structure. This structure is as follows:

```
struct input_info
{
  int  key_code ;                          // ASCII code of last character
  bool key_lshift ;                        // TRUE if left shift key pressed
  bool key_rshift ;                        // TRUE if right shift key pressed
  bool key_lctrl ;                         // TRUE if left ctrl key pressed
  bool key_rctrl ;                         // TRUE if right ctrl key pressed
  bool key_lalt ;                          // TRUE if left alt key pressed
  bool key_ralt ;                          // TRUE if right alt key pressed
  int  mouse_row ;                         // Row position of mouse's cursor
  int  mouse_col ;                         // Col position of mouse's cursor
}
```

One of 'getch' weaknesses is its inability to deal with the way extended keys are internally represented. Those are the keys that don't have an ASCII code associated with them. For example, the function, arrow and editing keys all return an extended keycode.

{tinput::get} will deal with these extended keys by adding 256 to the extended keycode. Appendix A lists all the extended keycodes currently available on an extended keyboard and their associated macros. Always use the macros instead of the real code.

The first parameter <ii> is a pointer to an {input_info} structure. Information is returned in this structure. You can set <ii> to a NULL pointer or call {tinput::get} without any parameter.

You then specify a <filter> to be used by {tinput::get}. A <filter> of 0 will allow any key to be read and returned by the function. You can also provide the ASCII or extended keycode (remember to add 256 to the real extended code or better yet, use the {II_*} macro) of the only key allowed to be returned by the function. This provides an easy way to WAIT FOR a specific key. The function will then return with that keycode, only when that key has been pressed.

➔ Beware of NEVER setting <filter> to an impossible key entry, or you will be trapped by the {get} function!

Next, you give the current context number <hlpctx> for this input. If the user presses F1, the right mouse's button or the on screen HELP button, the associated help screen for this context will be displayed. If you don't want to use the help system or if there isn't any help for the current context, set <hlpctx> to {EV_NOHLPCTX}.

See appendix C to learn how to build an help file and activate the help facilities.

➔ If you set <buffer> to {FALSE}, {tinput::get} will flush the keyboard buffer before reading a key. If set to {TRUE}, it will read from the keyboard and mouse buffers if keys have been previously pressed. By default, <buffer> is set to {FALSE}.

Return If a normal key was pressed, the returned value is an 'int' representing the ASCII code of that key (1-255).

If an extended key was pressed, the returned value is an 'int' representing the extended keycode plus 256 (256-396). See appendix A for a list of keyboard key macros.

If the mouse was pressed, the macro {EV_MOUSE} is returned.

The returned value is identical to the {key_code} field in the {input_info} structure.

{input_info}'s fields {mouse_row} and {mouse_col} will hold the cursor's coordinates at the moment the left button was pressed. They are undefined if the input wasn't a mouse event.

The other fields of the {input_info} structure hold the states of the shift, control and alt keys at the time of the input.

Example

```
input_info ii ;
if (input.get (&ii) == EV_MOUSE)
{
  // Process mouse input
}
else
{
  // Process keyboard input
}
```

C H A P T E R 15 : EasyVision's tmenubar class

The {tmenubar} class provides a menubar on the first line of the screen, with pull down menus and selection cursor.

A menubar acts as a filter. You make the user inputs go {through} the menubar. If the input is the keyboard event {II_F10} (F10), or an ALT-key corresponding to a menu hotkey, the menubar is activated. Otherwise, the input returns unchanged.

Items created in the menus are assigned return values. When the user selects an item, his input is changed to this return value and returned from the menubar.

➔ Keep in mind that all inputs going {through} and returned from the menubar are {input_info} structures. Refer to the chapter on the {tinput} class.

➔ An instance has already been globally declared and is called menubar. Only one instance of {tmenubar} can be used in a program. As a result, you will never declare and create another instance.

The {tmenubar} class has built in default values. However, if you would like to change those default behaviors, you can easily do so.

On the following pages, you will find each of {tmenubar}'s member functions.

Examples of using the menubar object are given in the source codes of the EasyVision's demo programs.

TMENUBAR::SETCOLORS

Summary Sets the background, foreground, highlight and cursor colors used by the menubar.

Syntax #include "tmenubar.hpp"

```
void far tmenubar::setcolors                          // Sets menubar colors
(
  int back=LIGHTGRAY,                                  // Background color
  int fore=BLACK,                                      // Foreground color
  int high=RED,                                        // Highlight color
  int cursor=GREEN                                     // Cursor color
) ;
```

Remarks You can use color macros if "CONIO.H" is included. Appendix B gives a description of available color codes and macros.

If you don't make a call to this function, the previously mentioned default values are assumed.

➔ When a menu has been created, it is thereafter illegal to change the already selected colors. Furthermore, the color 'DARKGRAY' is unavailable. It is reserved for offline menu items.

Return None.

Example

```
menubar.setcolors (BLUE,WHITE,RED,MAGENTA) ;
```

TMENUBAR::SETHLPCTX

Summary Sets the menubar's default help context number.

Syntax #include "tmenubar.hpp"

```
void far tmenubar::sethlpctx                    // Sets default context
(
  int hlpctx=EV_NOHLPCTX                        // Help context number
) ;
```

Remarks When created, each menu on the menubar and each item in a menu is given its own context number. If, however, a menu or an item is not given a number and help is requested while this menu or item is highlighted, the menubar default context number will be passed to the help routines.

Return None.

Example

```
menubar.sethlpctx (1000) ;
```

TMENUBAR::ADDMENU

Summary Adds a menu to the menubar.

Syntax #include "tmenubar.hpp"

```
void far tmenubar::addmenu                      // Creates a new menu
(
  char huge *name,                                  // Menu's name
  int      hotkey,                                 // Menu's hotkey
  char huge *sltext=NULL,                           // Statusline text
  int      hlpctx=EV_NOHLPCTX                     // Help context number
) ;
```

Remarks <name> is the name of the menu created. It can be of any length, but must fit on the menubar. The first two characters of the menubar are left blank, and the last 10 are reserved for the clock. Two spaces will be inserted between each menu.

There can be as many menus on the menubar as will fit. Just don't make the names too long.

The <hotkey> is the key that will activate the menu. It must be a letter (A-Z) or a digit (0-9), case insensitive. If the letter is present in the menu name, it will be highlighted. Two menus cannot have the same hotkey.

<sltext> is a short help text that will be displayed on the statusline when this menu is selected. It can be of any length.

<hlpctx> is the context number corresponding to this menu. The text associated to this context will be displayed if the user asks for help. If you set this argument to {EV_NOHLPCTX}, this menu won't have a help context of its own. Instead, the default menubar context will be use (see tmenubar::sethlpctx). See appendix 3 to learn about the help facilities.

Return None.

Example

```
menubar.addmenu ("Files",'F',"Opens a new file",100) ;
```

TMENUBAR::ADDITEM

Summary Adds an item to the last created menu.

Syntax #include "tmenubar.hpp"

```
void far tmenubar::additem                                  // Adds an item
(
  char huge *text=NULL,                                       // Item's text
  int       hotkey=II_NUL,                                  // Item's hotkey
  int       returnval=II_NUL,                                // Return value
  char huge *sltext=NULL,                                   // Statusline text
  int       hlpctx=EV_NOHLPCTX                          // Help context number
) ;
```

Remarks <text> is the text displayed for this menu entry. It can be of any length. The menu will autosize itself to accommodate the widest item.

There can be as many items in a menu as there is space on the screen. That is always 4 fewer than the total number of screen rows.

The <hotkey> is the key that will activate the item. It must be a letter (A-Z) or a digit (0-9), case insensitive. If the hotkey is present in the item's text, it will be highlighted. Two items cannot have the same hotkey.

If this item is selected, the user input will be change to <returnval> while in the {through} function. For example, if {II_A_X} (ALT-X) exits the program, you would set the QUIT item in the FILE menu to return the value {II_A_X} (Alt-X). It's a good idea to display those shortcuts in the item's text.

➔ This return value must be greater than 255. It cannot be {II_F10} (F10) or any of the arrows' keycodes (II_ARROW_*).

<sltext> is a short help text that will be displayed on the statusline when this item is selected. It can be of any length.

<hlpctx> is the context number corresponding to this item. The text associated to this context will be displayed if the user asks for help. If you set this argument to {EV_NOHLPCTX}, this menu won't have a help context of its own. Instead, the default menubar context will be use (see tmenubar::sethlpctx). See appendix 3 to learn about the help facilities.

➔ If {additem} is called with no arguments, it will insert a separator in the menu.

Return None.

Example

```
menubar.additem ("Open F3",'O',II_F3,"Open file",103) ;
```

TMENUBAR::ITEMSETAVAIL

Summary Sets availability of a menu item to {TRUE} or {FALSE}.

Syntax #include "tmenubar.hpp"

```
void far tmenubar::itemsetavail                    // Sets availability
(
  int  menuhotkey,                                 // Hotkey of menu
  int  itemhotkey,                                 // Hotkey of menu item
  bool state                                       // TRUE: available  FALSE: Not
) ;
```

Remarks You must provide the <menuhotkey> of the menu in which the item exists, and the <itemhotkey> of the item itself. Setting a menu item to {TRUE} will make it available, to {FALSE} not available.

➔ When a menu item is created, it is available by default.

Return None.

Example

```
menubar.itemsetavail ('F','O',FALSE) ;
```

TMENUBAR::THROUGH

Summary Makes an input go through the menubar.

Syntax #include "tmenubar.hpp"

```
input_info far tmenubar::through                    // Activates menubar
(
  input_info ii                                     // Input to go through
) ;
```

Remarks If <ii> contains the keyboard event {II_F10} (F10), the menubar is activated, but no menus are opened. If <ii> is a menu hotkey, this menu is opened.

Return If <ii> is not {II_F10} or a menu hotkey, {through} returns <ii> unchanged. If the menubar is activated, then if the user enters {II_ESC} (Esc), {through} returns with {II_NUL} in the {key_code} field of the {input_info} structure. Otherwise it returns with the return value of the menu item selected.

Example

```
input_info command ;
command = menubar.through (input.get (&command)) ;
```

TMENUBAR::REFRESH

Summary The menubar object has a {refresh} function that will redraw it on the screen.

Syntax #include "tmenubar.hpp"

```
void far tmenubar::refresh () ;                     // Redraws the menubar
```

Remarks You will most likely never call this function. You should call the desktop's own {refresh} function, who will then call this one.

Return None.

Example menubar.refresh () ;

C H A P T E R 16 : EasyVision's twindow class

The {twindow} class provides to your program powerful window facilities. Windows are where all your program's screen inputs and outputs will take place.

The {twindow} class comes with many member functions. They allow for manipulating the window position, size and attributes. Text can be easily written to the window. Input fields will give you formatted and secured user inputs. Push buttons will provide options selection. All these and many more features.

The {twindow} class has built in default values. However, if you would like to change those default behaviors, you can easily do so.

➔ Contrary to the other classes, instances of the {twindow} class have NOT been declare. You can use as many instances of {twindow} as you like. As a result, you will have to declare and create your own instances. First, you declare a pointer to a twindow object (twindow huge *win). Then you allocated memory for it (win = new twindow). The object has been initialised with default values and is now ready to be used.

➔ You can open simultaneously as many windows as you like. When you open a window, what was under it is saved to be restored when you'll close it.

➔ EasyVision doesn't keep track of the way windows overlap. It doesn't know if a window is over or under another one. Therefore, a window should NEVER be closed if another window covers part of it. Always close the ones that are in the foreground first.

➔ You CANNOT work with a window if it is under another one. If you do, the integrity of the desktop will be compromised.

On the following pages, you will find each of {twindow}'s member functions.

Examples of using window objects are given in the source codes of the EasyVision's demo programs.

TWINDOW::WINSETPOS

Summary Sets position of the next window to be opened.

Syntax #include "twindow.hpp"

```
void far twindow::winsetpos                    // Sets window's position
(
  int  row,                                     // Top left corner's row
  int  col,                                     // Top left corner's col
  bool movable=EV_MAYBE                         // TRUE: Can be moved
) ;
```

Remarks <row> and <col> are the topleft corner of the next window TO BE OPENED. The first and last lines of the desktop are reserved for the menubar and statusline. You can't have part of the window off the screen. Therefore, this function will validate all coordinates and change them to get a valid window position.

➔ If you set the size of the window before setting its position, this size will be taken into account to calculate the valid range of the <row> and <col> arguments.

➔ You CANNOT use this function once the window has been opened. Use the {winmove} function instead.

Return None.

Example

```
win->winsetpos (10,12) ;                        // Topleft corner at 10,12
```

TWINDOW::WINGETROW

Summary Returns row position of its topleft corner.

Syntax #include "twindow.hpp"

```
int far twindow::wingetrow () ;                 // Returns row pos
```

Remarks The top left corner of the screen is (1,1). But row one is unavailable because it is taken by the menubar.

Return Row position of window.

Example

```
row = win->wingetrow () ;
```

TWINDOW::WINGETCOL

Summary Returns window's column position of its topleft corner.

Syntax #include "twindow.hpp"

```
int far twindow::wingetcol () ;                    // Returns col pos
```

Remarks The top left corner of the screen is (1,1).

Return Column position of window.

Example

```
col = win->wingetcol () ;
```

TWINDOW::WINSETSIZE

Summary Sets size of next window to be opened.

Syntax #include "twindow.hpp"

```
void far twindow::winsetsize                       // Sets window's size
(
  int height=MAXINT,                    // Window's height including frames
  int width=MAXINT                       // Window's width including frames
) ;
```

Remarks <height> and <width> represent the size of the window, frames included. The first and last lines of the desktop are reserved for the menubar and statusline. Also, you can't have part of the window off the screen.

➔ This function will validate the asked size and change it to get a valid window size. The position that was set with {winsetpos} will be taken into account.

➔ You CANNOT change the size of a window once it has been opened.

Return None.

Example

```
win->winsetsize (10,60) ;                          // 10 rows by 60 cols
```

TWINDOW::WINGETHEIGHT

Summary Returns window's height in lines.

Syntax #include "twindow.hpp"

```
int far twindow::wingetheight () ;                 // Returns height
```

Remarks The top and bottom frames are included in the height.

Return The height of the window.

Example

```
heigth = win->wingetheight () ;
```

TWINDOW::WINGETWIDTH

Summary Returns window's width in columns.

Syntax #include "twindow.hpp"

```
int far twindow::wingetwidth () ;                    // Returns width
```

Remarks The left and right frames are included in the width.

Return The width of the window.

Example

```
width = win->wingetwidth () ;
```

TWINDOW::WINSETCOLORS

Summary Sets the background and foreground colors of a window.

Syntax #include "twindow.hpp"

```
void far twindow::winsetcolors                       // Sets window's colors
(
  int back=LIGHTGRAY,                                // Window's background color
  int fore=WHITE                                     // Window's foreground color
) ;
```

Remarks These are the colors used to draw the window. Those colors will be used by other functions when the color arguments are optional. <back> and <fore> are optional, and will default to 'LIGHTGRAY' and 'WHITE' respectively.

➔ You CANNOT change the default colors of a window once it has been opened.

➔ Bright background colors are available. See tdesktop's {settextmode}.

Return None.

Example

```
win->winsetcolors (BLUE,WHITE) ;
```

TWINDOW::WINSETTITLE

Summary Sets the title displayed on the top frame of a window.

Syntax #include "twindow.hpp"

```
void far twindow::winsettitle                    // Sets window's title
(
  char huge *title                               // Ptr to window's title
) ;
```

Remarks <title> can point to a title of any length, and only the portion that will fit on the top frame will be displayed. If you don't give a title before opening a window, no title will be displayed.

➔ You CAN'T set a title after a window has been opened.

Return None.

Example

```
win->winsettitle ("Open file") ;
```

TWINDOW::WINSETHLPCTX

Summary Sets the default help context for this window. If input fields or push buttons don't have a context of their own, they'll use this default.

Syntax #include "twindow.hpp"

```
void far twindow::winsethlpctx                   // Sets context's number
(
  int hlpctx=EV_NOHLPCTX                         // Help context's number
) ;
```

Remarks See appendix 3 for a description of the help system.

Return None.

Example

```
win->winsethlpctx (100) ;
```

TWINDOW::WINOPEN

Summary Opens a window.

Syntax #include "twindow.hpp"

void far twindow::winopen () ; // **Opens window**

Remarks The window is opened with default attributes, or the ones set by the previous functions. You CANNOT open an already opened window.

Default attributes for a window are:

Position is (row=2, col=1).
Size is (height=3, width=6).
Colors are 'WHITE' on 'LIGHTGRAY'.
The cursor is on line 1.

Return None.

Example

```
win->winopen () ;
```

TWINDOW::WINCLOSE

Summary Closes a window.

Syntax #include "twindow.hpp"

void far twindow::winclose () ; // **Closes window**

Remarks You CANNOT close an already closed window. When you close a window, all its attributes are reset to their default values as if you had just declared this object. Your previous settings like size and colors aren't in effect anymore.

All memory taken by the window is released. What was under this window when it was opened is restored.

➔ You should NEVER close a window that has part of itself hidden under another window. Always close the ones in the foreground first. This is your responsibility! EasyVision doesn't know your window's layer position and won't tell you if something goes wrong.

Return None.

Example

```
win->winclose () ;
```

TWINDOW::WINCLEAR

Summary Clears part or all the window content.

Syntax #include "twindow.hpp"

```
void far twindow::winclear                    // Clears an area in window
(
  int left=1,                                 // Top left corner
  int top=1,
  int right=MAXINT                            // Bottom right corner
  int bottom=MAXINT
) ;
```

Remarks The window must be opened to call this function. All arguments are validated and if incorrect, changed to fall within valid window coordinates.

All arguments are optional. Calling this function with no arguments will clear the entire window.

➔ Beware that this function will also clear buttons and input fields, BUT NOT REMOVE THEM. This will make them invisible, until you use them again. It is your responsibility to make sure you don't erase something important!

Return None.

Example

```
win->winclear (1,1,999,3) ;                   // Erases first 3 lines
```

TWINDOW::WINWRITE

Summary Writes text to the window. Many format options are available.

Syntax #include "twindow.hpp"

```
void far twindow::winwrite                    // Writes text to window
(
  char huge *text,                            // Ptr to text to be written
  int  row=EV_DEF,                            // Text's position in window
  int  col=1,
  int  format=0,                              // Justification
  int  fore=EV_DEF,                           // Text's color
  int  back=EV_DEF
) ;
```

Remarks The window must be opened. The <text> argument can point to a text string of any length, but only the first 132 characters will be considered.

➔ This function will only print on 1 line of the window. If the string is longer than an entire line, only what will fit will be printed. NO LINE WRAPPING WILL OCCUR with this function!

All arguments are optional, except for <text>.

<text> : Pointer to the text to be printed.

<row> : The {winwrite} function keeps track of the last line printed to, with an internal cursor. After each {winwrite}, the cursor is positioned on the next line.

When you use <row>, it tells where the text is to be printed. Row 1 is the first line under the top frame.

This argument is optional. If it is not given, or if you use the macro {EV_DEF}, {winwrite} will use it's internal cursor position. Text will be printed on the cursor line, and the cursor will move to the next line.

➔ If you DON'T give the <row> argument and write to the last window line, all the window will be scrolled up 1 line. YOU MUST MAKE SURE YOU DON'T HAVE ANY BUTTONS OR INPUT FIELDS. THEY WILL BE SCROLLED UP ALSO AND THE WINDOW'S INTEGRITY WILL HAVE BEEN COMPROMISED!

➔ If you give the <row> argument, you can then write to the last line and no scrolling will occur. The window cursor will stay on the last line.

<col> : Column where text will start. This argument is optional. If it is not given, it will default to 1.

<format> : Determines how the text string is justified.

0 : No justification. <col> is used.
1 : Left justified. <col> has no effect.
2 : Centered. <col> has no effect.
3 : Right justified. <col> has no effect.

This argument is optional. If it is not given, it will default to 0.

<fore> : Foreground color used. This argument is optional. If it is not given, or if you use the macro {EV_DEF}, it will default to the window's foreground color.

<back> : Background color used. This argument is optional. If it is not given, or if you use the macro {EV_DEF}, it will default to the window's background color.

Return None.

Example

```
                                                    // Writes to current line, left justified, current
                                                  // window's colors, and moves cursor to next line
win->winwrite ("Hello") ;

                                                 // Writes to current line, centered, current window's
                                                   // colors, and move cursor to next line.  Even if
                                                                  // <col> is 1, it has no effect.
win->winwrite ("Hello",EV_DEF,1,2) ;

                                                                 // Writes to specific line, specific column,
                                                                                  // specific colors
win->winwrite ("Hello",5,10,0,YELLOW,RED) ;
```

TWINDOW::WINTEXT

Summary Displays a text array in current window with word wrapping and 'Ok/Esc' prompts.

Syntax #include "twindow.hpp"

```
void far twindow::wintext                                          // Text with wrapping
(
  char huge *textptr,                                          // Ptr to text to be displayed
  int      fore=EV_DEF,                                         // Normal foreground color
  int      high=YELLOW                                             // Text's highlight color
) ;
```

Remarks The window must be opened to call this function.

<textptr> : This points to the text to be displayed. This text can be of any length.

This text is an array of characters that need not be formatted in any way. This function will display the array with word wrapping at the window's right edge.

Any extra spaces will be removed from the text, leaving only one space between each word. Any leading spaces to a line will also be removed. If you want to begin a line with spaces, or separate some words with more than one space, you must use the underscore (_) character.

You can highlight your text by surrounding characters with the tilde (~) character.

The array can be of any length.

An 'Ok' prompt will be created to allow the user to see the remaining text if it couldn't all fit in the window.

An 'Esc' prompt will be created to allow the user to stop viewing text at his convenience.

➔ When you use this function, make sure the window is totally empty. Everything that was in the window is erased when you call this function.

<fore> : Foreground color used. This argument is optional. If it is not given, or if you use the macro {EV_DEF}, the default foreground color of the window will be used.

<high> : Highlight color used. This argument is optional. If it is not given, the color YELLOW will be used for highlights.

Return None.

Example

```
win->wintext (instructions) ;
```

TWINDOW::WINTEXTFILE

Summary Displays a text file from disk in current window with word wrapping and 'Ok/Esc' prompts.

Syntax #include "twindow.hpp"

```
void far twindow::wintextfile                          // Displays a text file
(
  char huge *path,                                              // Path to file
  int     fore=EV_DEF,                          // Foreground color to use
  int     high=YELLOW                             // Highlight color to use
) ;
```

Remarks This function acts exactly the same as {wintext}. The only difference is that it gets it's input text from a disk file.

<path> : Complete path to disk file. If the file cannot be found, the string "File not found" is displayed instead. This string is pointed to by the system variable ev_filenotfoundtext.

<fore> : Foreground color used. This argument is optional. If it is not given, or if you use the macro {EV_DEF}, the default foreground color of the window will be used.

<high> : Highlight color used. This argument is optional. If it is not given, the color YELLOW will be used for highlights.

Return None.

Example

```
win->wintextfile ("C:\\autoexec.bat") ;
```

TWINDOW::WINMOVE

Summary Moves the windows to a new location.

Syntax #include "twindow.hpp"

```
void far twindow::winmove                    // Moves window to new position
(
  int row,                                   // Topleft corner's new position
  int col
) ;
```

Remarks The window must be opened. <row> and <col> are the new position of the topleft corner of the window. The first and last lines of the desktop are reserved for the menubar and statusline and you can't have part of the window off the screen. Therefore, this function will validate all coordinates and change them to get a valid window position.

Return None.

Example

```
win->winmove (10,12) ;
```

TWINDOW::WINSCROLL

Summary Moves window in 1 of 4 directions.

Syntax #include "twindow.hpp"

```
void far twindow::winscroll                  // Moves window 1 space only
(
  char direction                             // U: up  D: down  L: left  R: right
) ;
```

Remarks The entire window will be moved, IF POSSIBLE, one character in the requested direction. The name can be a little confusing. It is not the window content that is scrolled. It's the entire window.

The argument is a character, case insensitive:

U: Up, D: Down, L: Left, R: Right.

Return None.

Example

```
win->winscroll ('U') ;                       // Move window up one line
```

TWINDOW::WINONEDGES

Summary Checks if position is on window's edges.

Syntax #include "twindow.hpp"

```
bool far twindow::winonedges                    // Checks if on for edges
(
  int     row,                                  // Position
  int     col,
  int huge *offsetrow=NULL,                     // Offset to top left corner
  int huge *offsetcol=NULL
) ;
```

Remarks It will return {TRUE} if this position is on one of this window's edges. This function can also return the offset of the row and col positions to the upper left most corner of the window. If you don't want the offsets, set the return variables to 'NULL'.

Return {TRUE} if the position is on one of the edges.

Example

```
int off_row, off_col ;
if (win->winonedges (10,23,&off_row,&off_col))
  DoSomething () ;
```

TWINDOW::WININSIDE

Summary Checks if a position is in the window.

Syntax #include "twindow.hpp"

```
bool far twindow::wininside                     // Checks if in window
(
  int     row,                                  // Position
  int     col,
  int huge *offsetrow=NULL,                     // Offset to top left corner
  int huge *offsetcol=NULL
) ;
```

Remarks It will return {TRUE} if this position is inside the window. The frames are NOT considered inside. This function can also return the offset of the row and col positions to the upper left most corner of the window. If you don't want the offsets, set the return variables to 'NULL'.

Return {TRUE} if the position is inside window's edges.

Example

```
int off_row, off_col ;
if (win->wininside (10,23,&off_row,&off_col))
  DoSomething () ;
```

TWINDOW::WININPUT

Summary Makes window alive, allowing inputs to be made in multiple input fields and push buttons selected.

Syntax #include "twindow.hpp"

```
input_info far twindow::wininput                          // Gets input
(
  input_info ii                                           // input_info struct
) ;
```

Remarks The user will be able to move between input fields and buttons with {II_TAB} (Tab) and {II_S_TAB} (Shift Tab). He will be able to move the window, if allowed, by dragging it by one of its edges.

➔ You should refrain from using the {fieldinput} and {buttoninput} functions. Use this one instead.

Return If there are no input fields and no push buttons, the {input_info} structure is returned unchanged.

If no push buttons were created, but at least an input field, {wininput} will return with {II_CR} (Enter) or {II_ESC} (Esc). {II_CR} means the user confirmed the inputs by pressing ENTER. {II_ESC} means the user aborted the input by pressing ESC.

If buttons were created, {wininput} will return with the identification value of the button pushed.

Example

```
userinput = win->wininput (ii) ;
```

TWINDOW::FIELDSETCOLORS

Summary Sets the colors used in the next input field to be created.

Syntax #include "twindow.hpp"

```
void far twindow::fieldsetcolors                          // Fields' colors
(
  int back=BLUE,                                  // Fields' background color
  int foreon=WHITE,                             // Foreground color when active
  int foreoff=LIGHTCYAN                                 // Color when inactive
) ;
```

Remarks <back> : Background color used. This argument is optional. If it is not given, 'BLUE' is assumed. If the macro {EV_DEF} is used, it will default to the window default background color.

<foreon> : Foreground color used when the field is active. This argument is optional. If it is not given, it will default to 'WHITE'.

<foreoff> : Foreground color used when the field is inactive. This argument is optional. If it is not given, it will default to 'DARKGRAY'.

Return None.

Example

```
win->fieldsetcolors (GREEN,WHITE,WHITE) ;
```

TWINDOW::FIELDCREATE

Summary Creates a new input field in the window.

Syntax #include "twindow.hpp"

```
void far twindow::fieldcreate                             // Creates input field
(
  int      row,                                         // Input field's row
  int      col,                                         // Input field's col
  int      answerlength,                      // Answers' maximum length
  int      length,                          // Input field's length in window
  int      ftr,                           // Input field's filter number (0-4)
  char huge *xtraftr,                                 // Extra chars for filter
  bool     capsflag,                                // If TRUE, to uppercase
  bool     nullflag,                             // If TRUE, can enter null str
  char huge *defaultasw,                                  // Default answer
  char huge *sltext,                               // Statusline text for field
  int      hlpctx                                   // Help context number
) ;
```

Remarks The window must be opened before calling this function.

Fields have default built in behaviors when a window is first created. If you don't call any function to set their attributes, buttons will default to 'WHITE' on 'BLUE'.

<row> : Row of input field.

<col> : Column of input field.

<row> and <col> are validated to make sure the input field fits into the window. If the position is incorrect, it will be automatically changed.

<answerlength> : The maximum length of the input buffer. The user is allow to enter a string no longer than this limit. The upper limit is 32K. That should be enough!

<length> : The length of the input field in the window in characters. The input field cannot be wider than the window. The input field cannot be wider than the answer buffer's length.

<ftr> : An input field will allow only certain characters as input. <ftr> will determine what characters are accepted.

0 : All characters are allowed, except control chars.
1 : A-Z and a-z only. *** Space (32) not allowed ***
2 : 0-9 only.
3 : A-Z, a-z and 0-9 only.
4 : No characters allowed.

<xtraftr> : Include, between quotes, other characters that you want accepted by the filter. Often used to include the space char (ASCII 32).

<capsflag> : If set to {TRUE}, all inputs will be converted to CAPS.

<nullflag> : If set to {FALSE}, the user won't be allowed to input an empty string.

<defaultasw> : This is the default answer that will be put in the input field. This argument is optional. If you want the field to be initially empty, set to 'NULL'.

<sltext> : This is a short help text that will be displayed on the status line when the field is active. If you don't want a help text to be displayed, set to 'NULL'.

<hlpctx> : The help context number for this field. If set to {EV_NOHLPCTX}, the field will use the window's default.

Return None.

Example

```
win->fieldcreate (2, 2, 20, 10, 1, " ", TRUE, FALSE,
"Canada", "Enter country", 205) ;
```

TWINDOW::FIELDSETASW

Summary Sets a field current content.

Syntax #include "twindow.hpp"

```
void far twindow::fieldsetasw                          // Sets default answer
(
  char huge *answer,                                   // Default answer for this field
  int      fieldnb                                     // Field's ID number
) ;
```

Remarks Even if the content is immediately changed, the field on the screen will be updated only when it is activated.

<answer> : String to copy in the field. It can be of any length, but only what will fit in the answer buffer will be copied.

<fieldnb> : Number of the field to copy to. Number IDs are given to the field at their creation, starting with one.

Return None.

Example

```
win->fieldsetasw ("C:\\UTILS\\",5) ;
```

TWINDOW::FIELDGETASW

Summary Reads the content of the answer buffer of an input field.

Syntax #include "twindow.hpp"

```
void far twindow::fieldgetasw                          // Gets a field's answer
(
  char huge *dest,                                     // Destination to copy answer's to
  int      fieldnb                                     // Field's ID number
) ;
```

Remarks The answer buffer of field <fieldnb> is copied to <dest>.

➔ There is no way for the function to know if your destination is big enough to hold the content of the answer buffer. YOU MUST MAKE ABSOLUTELY SURE THAT YOUR DESTINATION IS AS BIG AS THE ANSWER BUFFER PLUS ONE CHARACTER for the terminating '\n'.

Return None.

Example

```
win->fieldgetasw (&response,3) ;
```

TWINDOW::FIELDINPUT

Summary Gets user input from one input field.

Syntax #include "twindow.hpp"

```
input_info far twindow::fieldinput                    // Activates field
(
  int fieldnb                                  // Field's number, to get input from
) ;
```

Remarks You must have created at least one input field to use this function. Fields are given numbers when they are created. The first field created is number one.

<fieldnb> : This is the field from which you will make the input. This field must exist.

➔ You should never have to use this function. You should consider using the {wininput} function instead. This is provided to allow for really special cases.

➔ The user won't be able to move the window.

Return {fieldinput} with return with an {input_info} structure containing the ASCII code representing how the user terminated the input. This can be {II_CR}, {II_ESC}, {II_TAB} or {II_S_TAB}.

Example

```
name = win->fieldinput (1) ;
```

TWINDOW::BUTTONSETCOLORS

Summary Sets the colors used for all the window's buttons.

Syntax #include "twindow.hpp"

```
void far twindow::buttonsetcolors                     // Sets buttons's colors
(
  int back=GREEN,                                  // Button's background color
  int foreon=WHITE,                          // Foreground color when active
  int foreoff=BLACK,                       // Foreground color when inactive
  int high=YELLOW                                  // Highlight color (hotkey)
) ;
```

Remarks All buttons use the same color configuration. You can't use this function once a button has been created.

<back> : Background color of all buttons. This argument is optional and will default to 'GREEN'.

<foreon> : Foreground color of active buttons. This argument is optional and will default to 'WHITE'.

<foreoff> : Foreground color of inactive buttons. This argument is optional and will default to 'BLACK'.

<high> : Highlight color of all buttons. This argument is optional and will default to 'YELLOW'.

Return None.

Example

```
win->buttonsetcolors (BLUE,WHITE,BLACK,RED) ;
```

TWINDOW::BUTTONCREATE

Summary Creates a new button.

Syntax #include "twindow.hpp"

```
void far twindow::buttoncreate                          // Creates a button
(
  int      row,                                         // Button's row
  int      col,                                         // Button's col
  char huge *name,                                      // Button's name
  int      buttonkey,                                   // Button's hotkey
  char huge *sltext=NULL,                               // Button's statusline text
  int      hlpctx=EV_NOHLPCTX                           // Help context number
) ;
```

Remarks The window must be opened to use this function. The window must be big enough to hold the button. It must be at least 4 lines high, and 11 columns wide.

You can create as many buttons as you want. There is no upper limit.

Buttons have default built in values when a window is first created. You don't need to call the {buttonsetcolors} function. If you don't, buttons will default to a 'GREEN' background, 'WHITE' text when active, 'BLACK' text when inactive, and 'YELLOW' highlight.

➔ The availability status of a newly created button always defaults to {TRUE} (available).

The first button created will be considered the default button. This button will be activated when you request a {buttoninput} or {wininput}.

<row> : Row position of the new button.

<col> : Column position of the new button. If <row> and <col> are not valid, they will be changed to a correct position.

➔ You must make sure buttons don't overlap. A button needs an empty line under and to the right of itself.

<name> : The name to put on the button. It can be of any length, but only the first 8 characters will be considered. Names aren't automatically centered on the buttons. Center the name manually by inserting leading spaces in the name.

<buttonkey> : This is the ASCII code that will identify a particular button. Some rules are to be observed:

If the identification code of the button matches a character in its name, that character will be highlighted.

{II_TAB} (Tab), {II_S_TAB} (Shift Tab) and {II_ARROW_*} (arrow keys) codes cannot be used to identify a button.

<sltext> : This is a short help text that will be displayed on the status line when the button is active. This argument is optional. If it is not given, no help text will be displayed.

<hlpctx> : The help context number for this button. If set to {EV_NOHLPCTX}, the field will use the window's default.

Return None.

Example

```
win->buttoncreate (10,2,"  Save",'S',"Save file",153) ;
```

TWINDOW::BUTTONSETAVAIL

Summary Sets the availability of a button.

Syntax #include "twindow.hpp"

```
void far twindow::buttonsetavail                    // Sets availability
(
  int  buttonkey,                                   // Button's hotkey
  bool available=TRUE                               // TRUE: available
) ;
```

Remarks The button specified MUST exist.

<buttonkey> : The button's hotkey that identifies the one you want to change.

<available> : {TRUE} means this button is available. {FALSE} means it is not. When a button is created with {buttoncreate}, its availability status is automatically set to {TRUE}.

Return None.

Example

```
win->buttonsetavail ('S',FALSE) ;                    // Save button OFF
```

TWINDOW::BUTTONINPUT

Summary Makes user go through the buttons with {II_TAB} (Tab), {II_S_TAB} (Shift Tab) and {II_ARROW_*} (arrow keys).

Syntax #include "twindow.hpp"

```
input_info far twindow::buttoninput                  // Waits for button
(
  input_info userinput,                              // User's initial input
  bool       first=TRUE                              // TRUE: Makes 1st alive
) ;
```

Remarks You must have created at least one button to call this function, and there must be at least one button available.

<userinput> : The initial user input, as an {input_info} structure. This could come from some previous processing. For instance, the result of an input field.

<first> : Determines which button will first be made active during the input. {TRUE} means the first created button, {FALSE} means the last. This argument is optional and will default to {TRUE}.

➔ You should never have to use this function. You should consider using the {wininput} function instead. This is provided to allow for really special cases.

➔ The user won't be able to move the window.

Return The function returns an 'int'. It is the identification code of the button that was pushed.

Example

```
command = win->buttoninput (command) ;
```

A P P E N D I X A : Keycodes macros

Extended keycodes are returned when you press a key that doesn't have an associated ASCII code. They are represented by stuffing two codes into the keyboard buffer. A 0 followed by an extended key keycode in the range 0 through 255.

The EasyVision's {getkey} function and {tinput} class, deal with these codes by returning values (int) in the range 0 through 511. The standard ASCII codes are returned unchanged (Guess why?). As a convenience, extended keycodes have 256 added to their real value, and are returned as a single number.

Macros, always debuting by {II_}, have been assigned to most of the values listed here. Those macros are available in "STDMACRO.H". You should always use the macros instead of the actual codes.

Macro	Value	Comment
II_NUL	0	/* No input at all */
II_MOUSE	256	/* {input_info} = mouse */
II_BS	8	/* Back space */
II_TAB	9	/* Tab */
II_CR	13	/* Carriage return */
II_ESC	27	/* Escape */
II_S_TAB	271	/* Shift-Tab */
II_A_Q	272	/* Alt-Q/W/E/R/T/Y/U/I/O/P */
II_A_W	273	
II_A_E	274	
II_A_R	275	
II_A_T	276	
II_A_Y	277	
II_A_U	278	
II_A_I	279	
II_A_O	280	
II_A_P	281	

II_A_A	286	/* Alt-A/S/D/F/G/H/J/K/L */
II_A_S	287	
II_A_D	288	
II_A_F	289	
II_A_G	290	
II_A_H	291	
II_A_J	292	
II_A_K	293	
II_A_L	294	
II_A_Z	300	/* Alt-Z/X/C/V/B/N/M */
II_A_X	301	
II_A_C	302	
II_A_V	303	
II_A_B	304	
II_A_N	305	
II_A_M	306	
II_F1	315	/* F1-F10 */
II_F2	316	
II_F3	317	
II_F4	318	
II_F5	319	
II_F6	320	
II_F7	321	
II_F8	322	
II_F9	323	
II_F10	324	
II_F11	389	/* F11 */
II_F12	390	/* F12 */
II_HOME	327	/* Cursor keys */
II_ARROWUP	328	
II_PAGEUP	329	
II_ARROWLEFT	331	
II_ARROWRIGHT	333	
II_END	335	
II_ARROWDOWN	336	
II_PAGEDOWN	337	
II_INS	338	
II_DEL	339	

II_S_F1	340	/* Shift-F1 to Shift-F10 */
II_S_F2	341	
II_S_F3	342	
II_S_F4	343	
II_S_F5	344	
II_S_F6	345	
II_S_F7	346	
II_S_F8	347	
II_S_F9	348	
II_S_F10	349	
II_S_F11	391	/* Shift-F11 */
II_S_F12	392	/* Shift-F12 */
II_C_F1	350	/* Ctrl-F1 to Ctrl-F10 */
II_C_F2	351	
II_C_F3	352	
II_C_F4	353	
II_C_F5	354	
II_C_F6	355	
II_C_F7	356	
II_C_F8	357	
II_C_F9	358	
II_C_F10	359	
II_C_F11	393	/* Ctrl-F11 */
II_C_F12	394	/* Ctrl-F12 */
II_A_F1	360	/* Alt-F1 to Alt-F10 */
II_A_F2	361	
II_A_F3	362	
II_A_F4	363	
II_A_F5	364	
II_A_F6	365	
II_A_F7	366	
II_A_F8	367	
II_A_F9	368	
II_A_F10	369	
II_A_F11	395	/* Alt-F11 */
II_A_F12	396	/* Alt-F12 */
II_C_PRTSCR	370	/* Ctrl-Print Screen */

II_C_ARROWLEFT	371	/* Ctrl- cursor keys */
II_C_ARROWRIGHT	372	
II_C_END	373	
II_C_PAGEDOWN	374	
II_C_HOME	375	
II_C_PAGEUP	388	
II_A_1	376	/* Alt-1/2/3/4/5/6/7/8/9/0/-/= */
II_A_2	377	
II_A_3	378	
II_A_4	379	
II_A_5	380	
II_A_6	381	
II_A_7	382	
II_A_8	383	
II_A_9	384	
II_A_0	385	
II_A_MINUS	386	
II_A_EQUAL	387	

A P P E N D I X B : Color codes and symbolic constants

When asked for a color argument, you must provide one of the following values. As an alternative, you can also use special macros, provided "CONIO.H" as been included.

➔ Functions that require a background color argument can use bright background color. Refer to {tdesktop}'s {settextmode} for more information.

Available background colors:

0 BLACK 8 DARKGRAY
1 BLUE 9 LIGHTBLUE
2 GREEN 10 LIGHTGREEN
3 CYAN 11 LIGHTCYAN
4 RED 12 LIGHTRED
5 MAGENTA 13 LIGHTMAGENTA
6 BROWN 14 YELLOW
7 LIGHTGRAY 15 WHITE

Available foreground colors:

0 BLACK 8 DARKGRAY
1 BLUE 9 LIGHTBLUE
2 GREEN 10 LIGHTGREEN
3 CYAN 11 LIGHTCYAN
4 RED 12 LIGHTRED
5 MAGENTA 13 LIGHTMAGENTA
6 BROWN 14 YELLOW
7 LIGHTGRAY 15 WHITE

A P P E N D I X C : Context sensitive help system

EasyVision provides an easy to use context sensitive help system. If fact, it's so easy you have nothing to do except write up the help text.

At startup, the static instance of the {tinput} class (input) will look in the directory of the executable file "*.EXE" for the help files "*.HLP" and "*.HDX". If those files are not found or one is missing, the program simply won't use the help system. When the user requests help by pressing the F1 function key, a message indicating that help is not available will be displayed.

If, however, you want your program to use the help system, follow these few simple steps...

What is a context

Menus in the menubar and items within a menu, each have their own context number. Windows, input fields and push buttons also have their own context number. They are given when you create those objects.

When one of these objects is active or selected, its context number is in effect. If help is requested by the user, the help text associated with this help context number will be displayed.

These objects provide easy input fonctions. You can also get inputs from the {tinput::get} member function. When you do, you also specify a current help context.

So, in essence, a context number indicates to the library what the user is currently doing in the interface or in the program. The help routines will thereafter be able to display help relative to the current situation.

Context numbering

You can number your context in any way. A context number is an integer number (int). You should consider, however, using a logical numbering system.

You could give a menu the context number 100, and give the items within this menu the numbers 101, 102, 103 and so on...

You would do the same with windows. A window would have the number 1000. The input fields would be 1100, 1101, 1102, etc. The push buttons could be 1200, 1201, 1202, etc.

You choose the numbering system that's right for you. You'll see in a few moments that there is an easier way to manage these context numbers.

Writing the ASCII help file

The help file is in plain ASCII. You can put comments everywhere in the file. The help compiler "HC.EXE" will consider only the text between the two keywords delimiters {HLPCTX} and {HLPEND}.

➔ The help text MUST be saved WITHOUT carriage returns or linefeeds at the end of the lines. You put a carriage return only at the end of a paragraph. If you want a blank line between paragraphs, put two returns at the end.

I strongly suggest that you write your help file with your favorite word processor and save the file in its native format. You'll also save an ASCII copy of it, without carriage returns at the end of the lines, for the help compiler.

You MUST give the ASCII file the same name as your program's executable "*.EXE". For instance, if the program is named "TOTO.EXE", your ASCII help file must be named "TOTO.*". The extension is up to you, but I strongly suggest you use "*.HLT" extensions for the ASCII help file "TOTO.HLT".

ASCII help file format

The ASCII help file uses the following format.

Example:

HLPCTX 123 MENU_EDIT_PASTE
By using the paste command you'll copy the content of the clipboard back onto the edit page at the cursor's location.
HLPEND

The help compiler will consider only what you put between the {HLPCTX} and the {HLPEND} keywords. You can put any comments you want elsewhere.

1. The HLPCTX keyword can be anywhere on the line and is case insensitive.

2. Next you put the context number. It must be separated from the {HLPCTX} keyword by at least one white space character (spaces (32) or TABS (9)) and ON THE SAME TEXT LINE.

   This context number must be a valid string that will evaluate to an integer.

3. After the context number, again separated by at least a white space character, and ON THE SAME LINE, you'll put the context '#define' macro.

➔ The compiler will produce a file with the name of your program executable and the extension "*.HCM" (Help Context Macros). You can then include this file in your source codes and reference easy to remember macros instead of context number.

This macro is optional. If not present, you'll be able to reference this context only by its number.

4. You then start your help text on the next line. Everything you write, up but not including the following {HLPEND} keyword, will be considered by the help compiler as the help text associated with this context number and macro.

5. You then put the terminating {HLPEND} keyword, wich is case insensitive. However, the preceding spaces will be considered part of the help text.

Take a look at the included examples that comes with the demo programs.

The help compiler

After your ASCII help file "*.HLT" as been saved, you compile it with EasyVision's help compiler "HC.EXE".

The command line is : HC [d:][\path\][filename.ext]

The compiler will parse the file checking for errors. If everything goes fine, it will produce two files. One with the extension "*.HLP", the other with "*.HDX".

The files will have the same name as your program's executable and be located in the same directory as the "*.HLT" file. As in our example, "TOTO.HLP" and "TOTO.HDX".

The "*.HLP" and "*.HDX" help files

The "*.HLP" and "*.HDX" files must be placed in your program's executable directory.

The "*.HLP" file is the compile help text. It stays on disk and is accessed as needed during execution. The "*.HDX" file is an index to the help file that is loaded in memory at program startup. This will provide extremely fast access to the help file, even if it's on floppy disk.

That's all there is to it! Your program can enjoy context sensitive help facilities with minimal work on your part.

Examples of creating and using the help system are given in the EasyVision's demo programs.

A P P E N D I X D : EasyVision's language system

EasyVision makes it easy to implement language files.

All the prompts and data strings output by your program should be placed in a separate file as global variables. You then reference them in the other modules with 'extern' declarations. This way, you can have different language versions of your application without rewriting a single line of code.

EasyVision displays a few default messages. Those are reference by the following system variables. You can change them manually to display appropriate messages for you country.

Two functions are also provided to switch the interface from french to english modes.

Language system variables

➔ All the following system variables are huge pointers to characters. The are reference in the header file "EVMSGS.HPP". You must include this file in every module that references these variables.

ev_helpwindowtitle

String displayed as the title of the help window.

English : "Help"

French : "Aide"

ev_helpwindownohelp

String displayed when an object does not have a help context or when the macro {EV_NOHLPCTX} is passed to {input::get}.

English : "Sorry, but help is not available at this time..."

French : "Desole, mais aucune aide n'est disponible en ce moment..."

ev_helpwindowfileerror

String displayed when the help system finds an error accessing the help files.

English : "There was a disk error while opening the help file. If the help files (*.hlp and *.hdx) are on a floppy disk, make sure it is still in the drive unit."

French : "Une erreur de lecture s'est produite en ouvrant le fichier d'aide. Si les fichiers d'aide (*.hlp et *.hdx) se trouvent sur disquette, soyez certain qu'elle est bien dans le lecteur."

ev_wintextdownbutton

In the {twindow::wintext} function and in the help system, string displayed on the button that will allow seeing the following page.

English : " Ok"

French : " Ok"

ev_wintextdown

In the {twindow::wintext} function and in the help system, string displayed on the statusline when the button that will allow seeing the following page is selected.

English : "See next page of text"

French : "Voir la prochaine page de texte"

ev_wintextquitbutton

In the {twindow::wintext} function and in the help system, string displayed on the button that will stop the display of text.

English : " Esc"

French : " Esc"

ev_wintextquit

In the {twindow::wintext} function and in the help system, string displayed on the statusline when the button that will allow stopping the following page is selected.

English : "Quit viewing text"

French : "Termine l'affichage du text"

ev_filenotfoundtext

In the {twindow::wintextfile} function, string displayed when the requested file is not found.

English : "File not found!"

French : "Fichier non trouve!"

ev_filetobig

In the {twindow::wintextfile} function, string displayed when the requested file is to big to be loaded in memory.

English : "Sorry, not enough memory to load this file..."

French : "Desole, pas assez de memoire pour charger ce fichier..."

ev_windowmove

String displayed when a window is moved by dragging one of its edges.

English : "Drag window while holding mouse's left button"

French : "Deplacez la fenetre en tenant le boutton gauche de la souris"

ev_statuslinehelp

String displayed in the left part of the statusline, indicating that the user should press F1 to activate the help system.

English : "Help"

French : "Help"

english() and french() functions

Two functions will allow you to reset the language system variables to their default values.

```
void language_english () ;                    // English definitions

void language_french () ;                     // French definitions
```

language_english () is automatically called at program startup. To call one of these functions, "EVMSGS.HPP" must be included.

A P P E N D I X E : EasyVision's demo program

I am a true believer in the motto 'An example will often be simple than its written explanation'.

I have included two complete demo programs with EasyVision. They are in the two archives "DEMO1.ZIP" and "HANOI.ZIP". The source codes, BC++ project files and executables are given.

I have tried to use as many functions as possible in these short programs. They are fully documented and they illustrate the working relationship between all those functions and classes.

I really think that 90% of your questions can be answered by going through these demo programs. You are invited to try some modifications of your own, and see the results.

I'll say it again... You should really print all this documentation for easier reading. It's also a good idea to print the header files for quick references.

Things to remember

A couple of things are to be remembered from this demo source code.

- You should always use 'huge' POINTERS in your code. You'll be on the safe side and avoid many problems.

- You should put all text and prompt strings in a separate resource file. This will make it easy to update prompts or make an alternate language file.

- EasyVision won't prevent you from using 'printf' statements. However, you should work within EasyVision's boundaries and use the provided output functions.

- All classes come with built in default values. Often, only one call is needed to use them if the defaults are to your likings.

Looking at this demo program, you can see that you can make great looking software faster than ever. Take the time to become familiar with EasyVision. The rewards will be fewer frustrations and more enjoyment out of your programming.

Have fun!

A P P E N D I X F : How to reach the author

I will gladly answer any questions relating to this software. I can be reach through the FidoNet or the Internet.

FidoNet : 1:240/1 (Make message to Remy Gendron)

InterNet : REMY_GENDRON@F1.N240.Z1.FIDONET.ORG

Phone : (418) 525-6803 (TNG SOFT's answering machine)

Mail : Remy Gendron
2480 ave de Vitre
Quebec, Quebec, Canada
G1J 4A6

I can't get back to you on the phone. Be sure to leave an electronic or conventional mail address. I'll get back to you real fast.

Any comments, bug reports or suggestions will be appreciated.

Thank you for considering this software!

Remy Gendron
Author of EasyVision

A P P E N D I X G : Trademarks

Turbo Vision from Borland.

Borland C++ 3.1 from Borland.

DeskViev from Quarterdeck.

CXL from Mike Smedley.

I N D E X

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W